An Qverview of Anna
A Specification Language for Ada

David Luckham & Friedrich W. von Henke

Technical Report No. 84-265
Program Analysis and Verification
Group Report No. 26

September 1984

This work was supported by the Advanced Research Projects Agency, Department of Defense, under contract N00039-84-C-0211.

An Overview of Anna, a Specification Language for Ada[1]

by

David C. Luckham
Friedrich W. von Henke

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

Abstract

A specification language permits information about various aspects of a program to be expressed in a precise machine processable form. This information is not normally part of the program itself.

Specification languages are viewed as evolving from modern high level programming languages. The first step in this evolution is cautious extension of the programming language. Some of the features of Anna, a specification language extending Ada, are discussed. The extensions include generalizations of constructs (such as type constraints) that are already in Ada, and new constructs for specifying subprograms, packages, exceptions, and contexts.

Anna has been designed in collaboration with B. Krieg-Brueckner and 0. Owe, see Anna 84, [2].

Key Words and Phrases: *Specification, Annotation, Formal Requirements, Software Methodology, Verification, Validation, Testing, Ada.*

This work was supported by the Advanced Research Projects Agency, Department of Defense, under contract NOO-039-82-C-0250,

[1] **Ada is a** registered trademark of the **U.S.** Department **of** Defense

1. Introduction

A specification language provides facilities for explaining a program. It permits the programmer to provide precise machine processable information about various aspects of a program that would not normally be part of the program itself. This may include, for exarnplc, the functional requirements for the prograrn — i.e., a mathematical description of what it is required to do — or properties of its components (variables, subprograms, modules) and how those components interact or depend on one another. Explanation may also contain background knowledge such as a description of the domain of values that the program operates on; such knowledge would be fundamental in constructing the prograrn and in checking or proving its correctness. The activity of explaining by providing information in some suitable language should normally be part of the process of designing and constructing programs. Explanations may serve as specification and thus precede implernentation of the program; other kinds of explanation may document aspects of the completed program. All kinds of explanation will help in debugging and maintaining programs.

Explanatory information is not, strictly speaking, necessary for computation although it may be expressed in executable form and it may facilitate compilation or optimization of compiled code.

Explanations must be given in a formalized language which can be mechanically processed and "understood" by a variety of support tools. There are essentially two approaches to designing specification languages: (1) the fresh start, and (2) the evolutionary approach, whereby an existing high-level programming language is extended. Both approaches have advantages. The fresh start does not have to accommodate the quirks of any given programming language design. The evolutionary approach is probably rnore likely to yield something that people try to use, and consequently build up the experience in use of formal specifications that is currently lacking. The Anna design is a result of the second approach, an extension of Ada to support the activity of explanation.

Ada itself already includes many constructs that are useful in explanation. For exarnple, in addition to strong typing, Ada introduces subtypes, derived types, packages, generic units, constraints, exceptions, and context specifications. These constructs play a dual role in the Ada design. On one hand they are intended to be useful in making both programs and compilation rnore efficient. On the other, they are intended to reduce programming errors by making programs rnore readable and by providing a great deal of error checking, at compile-tirne and at runtime. They improve readability by enabling the programrner to express programrning decisions explicitly. For example, variables of a given type whose values will be restricted may be declared with a subtype of that type; data structures with an intended set of legal operations can be defined by using the package and private type constructs; exceptional situations in which a computation should be terminated, and the processing that is intended to take place when they' occur, can be expressed using exception declarations and exception handlers.

However, from the point of view of explaining programs, and designs prior to irnplernentation, there are certainly some deficiencies. The explanatory role of some Ada constructs has been weakened by other considerations in the design and implementation of the language — a simpler or more restrictive construct often would have been preferable. For example, the rather permissive **Ada** visiblity rules allow sharing of global variables between different tasks without any formal declaration of intent to do so. As a result, although rendezvous constructs were intended to express task communication, the reader of a program cannot assume that they do. In other cases the explanatory

role of constructs has not been developed as much as it could have been, arid the programmer. must return once more to the good old informal comment to explain things. The rnost obvious example of this is the package declaration which provides only syntactic declaration of the facilities exported by the package; the semantics are left unspecified. In both of these kinds of situation, there are fairly straightforward extensions of the Ada constructs or additional constructs which wilt improve the explanatory capability of the language.

Overall, the Ada design seems to have incorporated a sufficiently rich base of explanatory constructs that specification and explanation of programs may be supported quite well — we do not yet know how well — within a fairly modest extension of the language. Some evidence in favor of this opinion can be drawn from the number of current efforts to develop Program Design Languages (PDLs) based on Ada (e.g., [16]). The question is what extensions would be most useful, and how to make the extensions so as to encourage their use.

In this paper we outline some of the features of a specification language for Ada called Anna (ANNotated Ada). Anna is designed as a language extension of Ada providing additional facilities for format specification and explanation. The extensions fall into three categories: (1) generalizations of explanatory constructs already in Ada, (2) addition of very obvious new kinds of constructs, mostly declarative in nature, dealing with exceptions, context clauses, and subprograms, and (3) addition of new specification constructs based on previous studies of the theory of program specifications, mainly to specify the semantics of packages and the use of composite types.

Our philosophy in the current Anna design has four main considerations.

1. Anna should be easy for an Ada programmer to learn and use. The syntax and semantics of Anna therefore remain as close as possible to the relevant Ada constructs. New predefined Anna concepts are introduced by means of standard Ada mechanisms, e.g., as new attributes and predefined operators and relations.

2. Anna should give the programmer the freedom to specify and annotate as much or as little as he wants and needs. In using Anna, there is no assumption that specifications have to be in any sense “complete”. The programmer is free to use only certain kinds of annotation, or to specify only a few features of the program. A wide range of Anna support tools providing error checking for different kinds of specifications is planned.

3. The current design should be a small extension of Ada that contains sufficient facilities to enable already established specification techniques to be applied to Ada programs — e.g., [3, 4, 5, 6, 8, 10, 13, 14, 15]. As experience with construction of specifications accumulates, those new concepts that prove to have wide application can be incorporated into future versions. This approach seems preferable to trying to provide a vast menu of special features based on several current specification methods. In fact, special facilities for tasking are not included in the present design since the subject of specification of concurrent computation is still very much a matter for research.

4. Anna should encourage developing new applications of formal specifications. Towards this goat, a significant part of the Anna language definition (see Anna84 [2], and also [9]) is devoted to defining a transformation of specifications into runtime checks. Consequently, an Anna program can be transformed into an Ada program with runtime checks for consistency with the original annotations and automatic reporting of

inconsistencies. This transformation can be implemented. The resulting ability to execute a prograrn against (or in comparison with) its formal specification provides a powerful toot for testing and debugging Ada programs. The Anna language definition also provides axiomatic semantics. This can be applied to verify Ada programs by mathematical proof of consistency between Ada text and its format Anna specification. Other possible applications which may influence future versions of Anna include rapid prototyping, automatic code generation from specifications, and the need for formal systems design languages.

This approach to specification language design attempts to "evolve" explanatory capabilities starting frorn a programming language that already contains useful explanatory constructs. The first step is to strengthen existing constructs and to add further ones where there is an obvious need. The approach perhaps errs on the side of caution in not introducing too many new concepts and constructs at once. As experience with program explanations develops, more explanatory features will probably be added. Simultaneously, we rnay hope that purely computational statements will disappear — possibly as we gain the ability to compile specifications into executable programs directly.

In the following sections we discuss some of the features of Anna and the ideas behind their design. The selection of topics is somewhat arbitrary and is not intended to cover the whole language. Our purpose is to give the reader an overview of Anna and a feeling for some of the problems and issues encountered in attempting this kind of extension of Ada. Many details are omitted or glossed over in order to focus on particular features of Anna; the reader is referred to Anna 84 [2] for the full picture.

2. Extension of Ada for Specification

An Anna program is an Ada program with formal comments. The syntax of forrnat comments is defined by extensions of the Ada syntactic categories and by sorne new Anna syntactic categories. There are some new predefined Anna attributes, operations, and logical operators, which may only appear in formal comments. From the point of view of Ada, formal comments are just comments; in an Anna program, however, the forrnat comments must obey the syntactic and semantic rules of Anna.

The decision to treat all annotations as Ada comments was made to ensure that Anna programs wilt be acceptable by standard Ada compilers and other Ada tools (which essentially ignore comments), in addition to special Anna toots.

There are two kinds of formal comments in Anna:

- **virtual** Ada **text:** each line of virtual Ada text begins with the symbol, -- :
- **annotations:** each tine of an annotation begins with the symbol, -- |

The Ada part of an Anna program is called the *underlying* Ada *program*. This consists of the Ada program itself — i.e., the part that is not comments. We often refer to parts of the underlying Ada program as *actual Ada text* to distinguish it from the virtual Ada text.

2.1 Virtual Ada Text

Virtual text is Ada text that is marked as virtual by the preceding virtual comment indicator.

There are many possible applications of virtual Ada text in specifying or annotating an actual program. The first intended use is simply to define programming concepts. Often the construction and explanation of a program will rely on concepts that are not explicitly implemented as part of the program. These "virtual" concepts must be precisely defined if they are to be used in any format specification. Most concepts can be formulated as mathematical functions or predicates (i.e., Boolean-valued functions). Anna therefore permits definition of virtual concepts by means of virtual Ada subprogram constructs.

Example: A virtual concept, LENGTH, for specifying a STACK package.

```
-- ITEM and MAX are assumed previously declared in the actual Ada text.
package STACK is

    --: function LENGTH return NATURAL:

        procedure PUSH(X : in ITEM);
            --| where in STACK.LENGTH < MAX,
            --|         out (STACK. LENGTH = in STACK. LENGTH+1);

        procedure POP (X : out ITEM) ;
            . . .
end STACK ;

package body STACK is
    type TABLE is array (POSITIVE RANGE <>) of ITEM:
    SPACE : TABLE(1 . . MAX):
    INDEX : NATURAL range 0 . . MAX := 0;

--: function LENGTH return NATURAL
    --| where return INDEX;   -- A result annotation defining the value of LENGTH.
    is separate:
            . . .
end STACK ;
```

In the example, the function, LENGTH, is not a part of the actual STACK package. It may be viewed as a concept the programmer used in formulating and designing the package. LENGTH is therefore delared as a virtual function. As such, it is not visible in the actual Ada text; it may only appear in virtual Ada text and annotations, e.g., to specify input and output conditions on PUSH. Notice that LENGTH is also declared with an annotation in the STACK body. It could bc used for runtime checking of correctness of calls to PUSH.

A virtual concept that has been introduced by an Ada subprogram specification can be defined either by annotations or by a virtual body. Anna does not require that bodies be supplied for virtual specifications. However, if a virtual body is given, the concept can be compiled and executed. Executable specifications obviousty provide a basis for various kinds of testing and validation of an actual program. It is of course expected that the virtual definitions are easier to write and understand, and less prone to error, than the actual text.

Virtual text may also be used to compute values that are not computed by the actual program, but are useful in explaining what it does. A typical example is a history sequence of values of an actual variable.

Restrictions on Virtual Text.
Virtual text must obey certain restrictions.

- First of all, it must be legal Ada. That is, it must obey the lexical, syntactic, and semantic rules of Ada if all the virtual text comment symbols, **"--:",** are deleted, with a few minor exceptions: it is allowed to contain some additional Anna attributes and operations, and bodies corresponding to declarations of program units may be omitted. The general implication of this is that the standard visibility rules etc. are satisfied.

- Secondly, it must not influence the computation of the underlying program by changing the values of actual objects. If this were allowed, it would not be possible to define the consistency of an Ada program with its Anna specifications since the specifications would alter its behavior. Anna therefore places additional restictions on virtual text so that any reference to an actual object is “read only”. These restrictions can be enforced by simple syntactic checks.

- Thirdly, virtual declarations may not hide entities declared in the underlying, actual Ada text. Without this restriction, the same name might have different meanings in the same context, depending on whether it appears in actual or virtual text.

2.2 Annotations

Annotations are built up from Boolean-valued expressions and reserved words indicating their kind and meaning. Anna provides different kinds of annotation, each associated with a particular language construct of Ada. These are annotations of objects, types or subtypes, statements, and subprograms; in addition there are axiomatic annotations of packages, propagation annotations of exceptions, and context annotations.

Any entity occurring in an annotation must be visible at that position in the Anna program. These constituent entities may be declared either in the actual Ada text or in virtual text (see for example the annotation of PUSH in the STACK package example). In some cases, the Ada visibility rules need minor extensions in order to ensure visiblity in annotations — e.g., so that formal parameters in subprogram specifications are visible in the subprogram annotations.

Every annotation has a region of Anna text over which it applies, called its scope. Its scope is determined by its position in the Anna program according to Ada scope rules. For example, if the annotation occurs in the position of a declaration, its scope extends from its position to the end of that declarative region. Essentially, most annotations are constraints on the values of program variables over their scope. The use of scope in defining the meaning of annotations has proved to be a powerful means of reducing the number of annotations from what would be required by the older assertional techniques [5, 15].

An annotation may be generic. This may happen when the annotation is part of a generic unit and contains formal generic parameters of the unit. In this case it is a template for annotations of instances of the unit. The Ada rules of instantiation of generic units apply to annotations.

Annotations may contain expressions whose values are not known until runtime. This permits annotations to have as general a form of parameterization as Ada provides for range constraints. As a result of permitting this generality, it is necessary for the Ada concept of elaboration to apply also to annotations. Anna defines the rules for elaboration of each kind of annotation. Elaboration or evaluation of an annotation is not permitted to have an effect on the Ada program (see restrictions on virtual text, 2.2). In particular, it is not permitted to change the state of the program, i.e. the value of any (actual or virtual) object. (Such a state change could happen if a function with side effect is called in an annotation.) Some other simple restrictions are assumed for expressions in annotations. If such restrictions are not assumed, the logic of annotations becomes unmanageable.

2.3 New Operations and Attributes

Concepts that prove to have wide application in explaining programs should be given standard definitions in Anna. This is directly analagous to those concepts that have proved to have wide application in constructing programs and are therefore predefined in Ada, either as operators, relations and attributes, or as predefined types. The Ada mechanisms for predefined concepts are used to provide the new predefined annotation concepts of Anna. We give two examples. (1) Ahna provides a new membership test, is in, in addition to the Ada membership test (the need for this is explained in 3.2). (2) The collection of objects designated by values of an access type, T, is an Anna attribute of the type T. The collection attribute, together with new associated operations of access types, is intended to facilitate specification of access type manipulations. Other predefined Anna operations and attributes are described in [2]. We expect that more operations or attributes may be added to future versions of Anna as the need for them is established.

2.4 Quantified Expressions

Anna extends the category of Ada expressions in several ways. The widest departure from Ada is the addition of the two quantifiers, for all and **exist** (and their negations) which we outline here. The meaning of quantifiers in Anna is more general than their meaning in classical first order logic because Anna expressions may not always have defined values.

The syntax of quantified expressions is as follows:

```
quantified-expression ::=
    quantifier domain {; domain} => boolean_compound_expression

domain ::=
    identifier-list : subtype-indication

quantifier ::=
    [not] for all | [not] exist
```

The identifiers in the domain of a quantifier are called logical *variables.* They are declared by the quantifier, so the scope of a logical variable extends from its occurrence in the domain to the end of the (boolean) compound expression following the symbol, **=>.** The meaning of quantified (boolean) expressions is as follows:

for all X : T => P(X) *means* "for all values X of (sub)type T if P(X) is defined then P(X) is true".

exist X : T => P(X) *means* "there exists a value X of (sub)type T such that P(X) is defined and true".

Thus, in Anna a quantified expression will have a defined value even if P(X) is undefined for some value in T. If P(X) is defined for every value X in T, then the Anna meaning of quantifiers coincides with their meaning in classical logic. In fact, **for all X** : T **=>** P(X) is true if and only if there exists no value X in T for which P(X) is both defined and false. Thus the generalized interpretation of quantification in Anna preserves the standard relationship between the quantifiers, i.e.,

for all X : T => P(X) *is equivalent to* **not exist X : T => not P(X).**

Examples of quantified expressions:

```
type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);

for all X : DAY => X'LENGTH = 3;
     -- This is true because the boolean expression after = > is true
     -- for each of the seven string values in the enumeration type, DAY.

for all X : DAY => exist P : PERSON-RECORD => P.BIRTHDATE.DAY = X;
     -- Suppose the type, PERSON_RECORD has a BIRTHDATE component which has a
     -- DAY component. This quantified expression is true since for any of the seven
     -- values of DAY there is a record aggregate of type PERSON-RECORD with a
     -- BIRTHDATE component having that value as DAY component.

for all N : NATURAL => exist S : NATURAL => S ≤ SQUARE-ROOT(N) < S+1;
```

The introduction of quantifiers provides an annotation language that is as rich as classical first order logic. Careful use of quantifiers results in concise and readable annotations. However, establishing consistency of Ada text with annotations involving quantified expressions generally requires mathematical proof; the alternative of runtime checking (see Section 5) is cornputationally practical only if the domains of the quantifiers are small. For example, the first annotation, **for** all X : DAY = > **X'LENGTH =** 3, is easily checkable (since it does not contain program variables it can be checked at compile time). The second annotation could be established by constructing seven aggregate values, but this would require a very smart "checking method". The third annotation is best established by mathematical proof.

3. Kinds of Annotations

In this section we give examples of different kinds of annotations and their meanings.

We note first that an annotation may contain two kinds of variables: *logical variables* declared in the annotation (for instance by quantifiers), and *program variables* declared in Ada text that are visible at the position of the annotation.

Secondly, the meaning of most kinds of annotation is defined in terms of the sets of *observable program states* of computations of their scope. A program state associates a value with each program variable at a point during a computation. An *observable state* is one that results from the execution of a simple statement or the elaboration of a declaration. (Note that this would not include, for example, states during execution of a subprogram call; they are not observable from the scope containing the call since the call is a simple statement.) A set of states of a scope consists of the *initial state* resulting from elaboration of the scope, the observable states resulting from the initial state by elaboration of declarations and execution of simple statements within the scope, and the *final state* when the computation in the scope terminates normally. An annotation, generally speaking, constrains all sets of observable states of its scope.

3.1 Object Annotations

An object annotation is a boolean expression occuring in a declarative part and containing one or more program variables (which must, of course, be visible at that point). Such an annotation is generally an *object constraint;* it constrains the values of its program variables within its scope.

Examples of object annotations:

```
X : INTEGER := E; --| X <= BOUND(E);
-- Values of X must be less than the value, BOUND(E) throughout the scope of the annotation,
-- which, in this case, "covers" the scope of X.

M, N : INTEGER := 0;      --| N ≤ M;

-- The values of M and N are constrained so that N is less than or equal to M in any
-- observable state throughout the scope of the annotation.
```

Object annotations provide a simple means of specifying the values of objects within declarative regions. They are a more powerful form of the old-style assertions [5, 15). In fact, an object annotation is equivalent to a set of assertions, one placed at each position in its scope where one of its program variables might change value. The annotations of the same object may differ from region to region.

3.2 Type and Subtype Annotations

A subtype annotation is used to constrain a type or subtype. This kind of annotation is essentially a straightforward generalization of the Ada range constraint concept. A subtype annotation follows immediately the type or subtype declaration and is bound to it by the reserved word, where.

A subtype annotation restricts the doniain of values of the (sub)type. Thus in,

```
subtype S is T;   --I where X : S => C(X):
```

where C(X) is a boolean express&, the values in S are restricted to the set of all values X of type T that satisfy C(X); the expression C(X) is also called the *subtype constraint*. Since the domain of the subtype is restricted, this implies that the values of any object or variable declared of that subtype must satisfy the constraint — e.g., constants and program variables, parameters, generic parameters and logical variables in quantified expressions.

We note that **where** not only serves to bind the annotation to the type declaration but also declares X; the X in the annotation is a logical variable. Any program variable occurring in a type annotation is a parameter of the annotation; this means that it designates its value when the annotation is elaborated.

Examples of subtype annotations:

```
subtype EVEN is INTEGER:
    --I where X : EVEN => X mod 2 = 0;
    -- The constraint on the type EVEN picks out the even values from the set of integers.
    -- Since this is not a range, the constraint cannot be expressed as an Ada constraint.

type TABLE is array (NATURAL range <>) of ITEM;
type QUEUE is
    record
        STORE      : TABLE(0 . . SIZE-l);
        COUNT      : NATURAL range 0 . . SIZE : = 0;
        IN-INDEX   : NATURAL range 0 . . SIZE-l := 0:
        OUT-INDEX  : NATURAL range 0 . . SIZE-l := 0;
    end record ;
    --| where Q : QUEUE =>
    --|     Q.IN_INDEX = (Q.OUT_INDEX+Q.COUNT) mod SIZE:
    -- This constraint on the record type cannot be expressed in Ada.
```

Subtype annotations generalize the Ada range constraint and can express more subtle properties since the Anna constraint can be any Boolean expression. However, this added power has repercussions on the Anna language design. We discuss two of them briefly here.

The first is the need to introduce a new membership test, **is in** for use in annotations. The Ada test, **in,** is evaluated without regard for Anna type annotations (which are merely comments as far as Ada is concerned); therefore it may yield the value TRUE in cases where the Anna constraint is FALSE. For example, within the scope of the subtype EVEN (above), the test, 3 **in** EVEN, has the value TRUE. In such cases the Ada test is inconsistent with the annotation (see Section 5). However, if a membership test is used in an annotation, it should definitely be consistent with any (sub)type annotation, and therefore has to be different from the Ada test. Therefore Anna introduces a new membership test, **is in,** which is evaluated in conjunction with type annotations, e.g., 3 **is in** EVEN has the value FALSE.

Secondly, there is the problem of updating the value of a composite object consistently with a type annotation. Normally, the relevant components of the object would be assigned new values separately; the final value may satisfy the type constraint while the intermediate values do not. For example, in assigning a new value to an object of type QUEUE (above), the IN-INDEX and COUNT components may be changed separately while the OUT-INDEX remains unchanged. The only

satisfactory solution is to provide for a "hole" in the scope of the type declaration where the annotation is not required to hold. Fortunately, it is not necessary to provide for "holes" in some arbitrary manner; the boundary between an Ada package body and the outer scope defines such a "hole". A *modified* type constraint can be placed on a type that is declared in a package specification. Such a constraint is not required to hold *inside* the body of the package, but only upon exit from the body (see [2]). This relaxed form of a type constraint has exactly the semantics of Hoare's monitor invariant [7].

3.3 Statement Annotations

Statement annotations are used to specify properties of statements. They are annotations appearing in a sequence of statements, and they have as scope either the immediately preceding or succeeding statement depending on their kind (simple or compound annotation). A simple statement annotation constrains the state after execution of the preceding statement. A compound statement annotation constrains the execution of the succeeding compound statement.

Examples of simple annotations of statements:

```
X := X+1;    --| X = in X+1;
     -- The final value of X must be equal to the initial value incremented by 1.

if A(X) > A(X+1) then
     Y      := A(X+1);
     A(X+1) := A(X);
     A(X)   := Y;
end if;
--| A(X) ≤ A(X+1);
-- after execution of the preceding compound statement, the components A(X)
-- and A(X + 7) will be in the specified order.
```

Statement annotations may be used to express simple kinds of program specifications such as assertions and loop invariants [5, 15]. (For examples of compound statement annotations see [2].)

3.4 Subprogram Annotations

Subprogram annotations are annotations associated with Ada subprogram specifications by the reserved word, **where.** Subprogram annotations extend the Ada specification part and permit specification of a subprogram independently of the body that implements it. Such annotations may include constraints on the formal parameters, results of function calls, and conditions under which exceptions may be propagated. Subprogram bodies may also be annotated using subprogram annotations, and in addition may contain other kinds of annotation within the body after the reserved word, is.

A subprogram annotation must be true of every call to the subprogram (and therefore constrains all calls) and must be correctly irnplernented by the body (and therefore constrains the declarative region of the subprogram body).

Examples of subprogram annotations:

```
function "/" (NUMERATOR, DENOMINATOR : INTEGER) return INTEGER;
    --I where DENOMINATOR ≠ 0:
    -- The actual value of DENOMINATOR has to be different from 0. This places
    -- a constraint on all calls to "/".

function COMMON-PRIME(M, N : INTEGER) return NATURAL:
    --I where return P : NATURAL =>
    --|       IS-PRIME(P) and M mod P = 0 and N mod P = 0;
    -- A result annotation defining the value returned; see [Anna 84.]

procedure SWAP(U, V : in out ELEM);
    --I where out (U = in V and V = in U);
    -- The modifiers, in and out refer to initial and final values of the parameters.
```

3.5 Exception Propagation Annotations

Specification of exceptional behavior requires annotation of exception handlers, **raise** statements, and units which may propagate exceptions (see, e.g., [11, 12]). The first two requirements may be achieved using statement annotations. The last however requires introduction of a new kind of annotation, called *propagation annotations,* for specifying the conditions under which an exception may be propagated.

There are two kinds of propagation annotation, weak and strong: they have syntactic forms as follows:

```
weak-propagation-annotation ::=
    raise exception-choice {| exception-choice}
        [=> compound-expression]

strong-propagation-annotation ::=
    compound-expression => raise exception_name
```

These annotations may appear as subprogram annotations (i.e., as annotations of the specification part), as statement annotations, and as annotations over a declarative region. They are mostly used in specifying subprograms.

A *weak* propagation annotation of a subprogram specifies exceptions that may be raised as the result of a call to that subprogram, and a condition that will be satisfied by the state of the calling environment when the exceptions are propagated. That is, the boolean expression subsequent to => will be true of the computation state of the calling environment at the point when the execution of the subprogram body is abandoned.

A *strong* propagation annotation on a subprogram specifies a condition on the **in** values of parameters (and global variables) of a call under which an exception must be propagated; the condition need not be true after propagation.

The difference between the two kinds of annotation may be described as follows. A weak annotation informs the user what to expect if a call propagates an exception, and thus enables construction of an

outer handler. A strong annotation specifies to an implementor what input conditions must result in propagation of an exception. Historically, the inclusion of both kinds of annotation was motivated by the axiomatic semantics which has been defined only for the weak annotations. However, it turns out that it is useful to have both kinds available in specifications.

A propagation annotation may also be given without any condition; in this case it documents in the subprogram specification that an exception may be propagated from the body.

Examples of propagation annotations:

```
procedure BINARY-SEARCH(A        : in ARRAY-OF-INTEGER;
                        KEY      : in INTEGER:
                        POSITION : out INTEGER );
--| where ORDERED (A),
--|       out (A(POSITION) = KEY),
--|       raise NOT-FOUND => for all I in A'RANGE => KEY ≠ A( I);
-- A weak propagation annotation: whenever NOT-FOUND is propagated and A is ordered
-- then KEY is not a component of A.

procedure PUSH(E : in ITEM);
    --j where in STACK.LENGTH = SIZE => raise OVERFLOW,
    --|       raise OVERFLOW            => STACK = in STACK;
    -- A combination of both weak and strong propagation annotations: whenever the stack
    -- length is equal to SIZE, a call to PUSH must propagate OVERFLOW; whenever
    -- OVERFLOW is propagated the stack state will be equal to its input value.

X := Y div Z; --I raise NUMERIC-ERROR => Z = 0 and X = in X:
```

3.6 Context Annotations

Context annotations provide a facility for specifying the use of outside variables within a program unit, and also the use of variables from an Ada context within a compilation unit. The syntax is:

```
context-annotation : := limited [to name_list;]
```

where the name list is a list of variable names. A context annotation immediately precedes a unit. It specifies that only the variables (from the enclosing scope or compilation context) occurring in the name list are referenced within the unit; if the name list is empty then no outside variables are referenced in the unit.

Examples:

```
-- | limited
    package body STACK is
        ...   -- No outside variables are referenced in the STACK body.
    end STACK;


-- | limited to SIZE;   -- No global variables except SIZE may be used in this body.
package body QUEUE is
```

```
    type TABLE is array (NATURAL range <>) of ITEM:

    STORE      : TABLE(0 . . SIZE - 1);
    COUNT      : NATURAL range 0 . . SIZE := 0;
    IN-INDEX   : NATURAL range 0 . . SIZE -1 := 0;
    OUT-INDEX  : NATURAL range 0 . . SIZE -1 := 0;

     -- | limited to COUNT:   -- Of all variables visible at this point, only COUNT may
    function EMPTY return BOOLEAN is   -- be used.
    begin
        return COUNT = 0;
    end EMPTY;

    -- | limited to STORE, COUNT, IN-INDEX:
    -- The variables OUT_INDEX and SIZE may not be used in ADD.
    procedure ADD( X : ITEM) is separate;
    ...
end QUEUE;

separate (QUEUE)
    -- | limited to STORE, COUNT, IN-INDEX;
    -- The context annotation preceding the body stub is repeated here.
procedure ADD(X : ITEM) is
begin
    ...   -- Only the variables STORE, COUNT, IN-INDEX may be referenced.
end ADD:
```

It is very likely that context annotations in future revisions of Anna will be extended to permit naming of not only outside variables, but also subprograms and packages.

4. Package Annotations

In this section we describe some features of Anna intended for specifying packages. We shall deal mainly with some of the facilities for specifying a package declaration.

The need for formal methods of explaining Ada packages is overwhelming. Packages are intended to provide a clear separation between information that is available to a user and internal implementation details that should be hidden from users. This separation should allow one implementation to be replaced by another without any change to a using program being necessary.

However, Ada package declarations contain only syntactic information about the visible entities of the package. The only way to describe the behavior of these entities independently of the body is by comments placed in the specification part. Informal comments are highly unsatisfactory both as a specification to be implemented, and as documentation for the potential user. They do not provide a basis for any kind of autornated aid, e.g., automated error checking. Furthermore, since they are not even subject to the visibility rules of the language, they may mention implementation details of the package body, thus destroying the separation they were intended to maintain.

4.1 Visible and Hidden Annotations

Any kind of formal comment can appear in a package declaration or body including the previously described kinds of annotations. The meaning of annotations of entities in a package is the same as before; their region of visibility conforms to the Ada rules for packages. Annotations in the visible part of a package declaration are called *visible annotations;* their visibility is the same **as** an Ada declaration at the same position. Annotations in the private part and body are called *hidden annotations;* the visibility of hidden annotations is restricted to the private part and body.

The purpose of visible annotations is to specify the package data types and subprograms. Such specification cannot refer to the body of the package (a simple consequence of Ada visibility rules). Visible specifications should be sufficient to understand how to use a package without inspecting its implementation in the private part and body. They may also serve as a specification for implementing the body.

Hidden annotations are used to specify the intended behavior of the implementation of the package. They may also define the virtual functions declared in the visible part (for use in the visible annotations) in terms of the local items in the private part and body (see, e.g., the STACK example in 2.1).

Although visible annotations cannot refer explicitly to hidden entities in the package body, they logically imply constraints on the body which must be satisfied if the body is to be consistent with them. The implied constraints can be computed from the visible annotations; the details are given in Anna 84 [2].

Examples of package declarations with visible annotations.

```
package COUNTERS is

    type COUNTER is limited private ;
--: function VALUE (C : COUNTER) return NATURAL;

    procedure INITIALIZE(C : COUNTER):
    --| where out (VALUE(C) = 0):

    procedure INCREMENT( C : in out COUNTER) ;
    --| where out (VALUE(C) = VALUE(in C) + 1);

private
    ...
end COUNTERS;
```

COUNTERS is specified in terms of a virtual mapping, VA LUE, from values of type COUNTER into the natural numbers. Any implementation is correct if there is a definition of VALUE such that the annotations of INITIALIZE and INCREMENT are satisfied. Details of implementations (which could be complex, depending on machine word length, etc.) are left open except for requiring such a mapping. Note that VALUE is not required to be one — one. This property can be expressed by axiomatic annotations (below).

```
package RECOGNIZER is

    --: type STATE is (SO, S1, S2, S3);
    --: S : STATE := SO;

        type BIT is INTEGER range 0 . . 1;

        procedure INITIALIZE;
        --I where out S = SO:

        procedure NEXT-BIT(X : in BIT: Y : out BOOLEAN):
        --| where
        --| out (if in S=S3 and X=0 then Y=TRUE else Y=FALSE end if),
        --| out (if in S = SO then
        a-|           if X = 0 then S = S1 else S = SO end if
        --|       elsif in S = S1 then
        --|           if X = 1 then S = S2 else S = S1 end if
        --|       elsif in S = S2 then
        --|           if X = 1 then S = S3 else S = S1 end if
        --|       else
        --|           S = SO
        --|       end if ) ;

end RECOGNIZER;
```

The package, RECOGNIZER, is specified by a virtual finite state machine with four states. S is declared as a global variable of the two procedures, and the state transitions are specified by changes in the value of S. From this specification it is easily seen that NEXT-BIT will recognize (i.e., return the value TRUE for Y) sequences of values of X of the form 0, 1, 1, 0. The package body may implement the states in a variety of ways (e.g. as pairs of binary counters) but the details should not affect the outside user. Obviously, this specification could be more precise if we had the ability to "talk about" the state of the package itself. We would then be able to identify the state of the virtual machine with the states of the package.

It is often not possible to specify the visible part of a package adequately using only the previously described kinds of annotations for objects, types and subprograms. For example, it is not possible to state algebraic relationships between package functions. There are many examples of packages where the most natural and complete description uses algebraic specifications. The need for some more powerful kinds of visible annotations is therefore clear. Anna introduces two new kinds of annotation concepts for use in specification of package declarations: *package states* and *axioms.*

4.2 Package States

Viewed from the outside, a package is treated conceptually as an object of some new type (not defined in Ada). The values of this type are called *states.* For each package the *type* of its states and the values of its *initial state* and *current state* are attributes of that package. Thus for a package, P, P'TYPE, P'INITIAL and P'STATE designate the type of its states, the initial state and the current state respectively. For convenience, the name of a package, P, may be used to denote its current state in annotations.

The concept of state contains the following premises:

- The structure of a package state is not visible from the outside.
- A state may change only as the result of a call to a visible subprogram of the package. (A state change can occur even if a subprogram call does not terminate normally — e.g. propagates an exception.)

Thus, viewed from the outside, the only information available to differentiate one state from another is the sequence of calls to visible subprograms that has been executed. Therefore such sequences must be expressible in annotations. We note that previous work on the specification of module constructs has advocated the use of sequences of operations, see e.g. [3, 10, 14]. States resulting from the execution of a sequence of subprogram calls, each of which terminates normally, are called *successor states.* Successor states are denoted in Anna by sequences of package operations and are names in Anna. The notation for successor states has been chosen so that these expressions are easily distinguished from any other kind of expression:

```
package-state : := state-name[subprogram-call{; subprogram_cal 1}]
```

An example of a successor state:

```
STACK[PUSH(X); POP(Y)]
-- Designates the state resulting from any state of the STACK package when successive
-- calls, PUSH(X), POP(Y) terrninate normally.
```

As a consequence of the second premiss, a package state may not depend on values of visible or global objects. If it did, operations other than calls to package subprograms could change its state. Anna does not provide the ability to describe the states of such packages explicitly. The state concept is intended for specification of those packages that are constructed so that all components of the state are inside the package hidden part — this implies that the context annotation, **limited,** which excludes global variables, should be applicable to the package.

Finally, inside the package body, its state has the structure of a record whose components are the values of objects local to the body; here, “object” is used in a more general sense to include the states of local packages. Thus, the package state type has a semantics similar to a lirnited private type declared that the beginning of the package specification, and having a full declaration as a record in the package body. The equality operator, " = ", is defined in Anna for state types.

Example: specification for RECOGNIZER using the state type:

```
package RECOGNIZER is

   --: so, S1, s2, s3 : constant RECOGNIZER'TYPE;

       type BIT is INTEGER range 0 . . 1;

       procedure INITIALIZE;
       --I where out RECOGNIZER = SO;

       procedure NEXT_BIT(X : in BIT; Y : out BOOLEAN);
       --I where
       --I out (if in RECOGNIZER=S3 and X=0 then Y=TRUE else Y=FALSE end if),
```

```
--| out (if in RECOGNIZER = SO then
--|          if X = 0 then RECOGNIZER = S1 else RECOGNIZER = SO end if
--|     elsif in RECOGNIZER = S1 then
--|          if X = 1 then RECOGNIZER = S2 else RECOGNIZER = S1 end if
--|     elsif in RECOGNIZER = S2 then
--|          if X = 1 then RECOGNIZER = S3 else RECOGNIZER = S1 end if
--|     else
--|          RECOGNIZER = SO
--|     end if);

end RECOGNIZER;

-- In the annotations, RECOGNIZER, designates the current state of the package.
-- This specification relates the package state type directly to the state transitions
-- and is therefore more precise than the previous example.

RECOGNIZER[INITIALIZE; NEXT_BIT(A, Y); NEXT_BIT(B, Y)]
-- If A = 0 and B = 1 this successor state designates the state S2.
```

4.3 Package Axioms

Axioms are annotations declared within the visible part of a package after the reserved word **axiom.** They express properties of the visible entities of a package that are guaranteed by the package body. The meaning of package axioms is that (i) they are visible promises that may be assumed wherever the package specification is visible, and *(ii)* they are constraints on the hidden part of the package. As visible promises, package axioms express properties of the visible entities of the package that are promised to hold in the scope of its declaration. For example, in analyzing the correctness of a program that uses a package, the package axioms may be assumed. In the package hidden part each axiom corresponds to an assertion on the local entities. These hidden assertions may be constructed from the axioms according to rules given in [2]. If the implementation of the body satisfies the assertions then the axioms in the visible part of the package are consistent with the package body.

Axioms contain only logical variables after elaboration; i.e., all program variables are treated **as** parameters and are replaced by their values at elaboration.

Example: A more complete specification of COUNTERS:

```
package COUNTERS is

        type COUNTER is limited private:

    -- : function VALUE (C : COUNTER) return NATURAL:
    --| axiom for all C, D : COUNTER; N : NATURAL =>
    --|       VALUE(C) = N and VALUE(D) = N → C = D;

    ...  -- Remaining specifications as in previous example.
end COUNTERS;
-- The mapping, VALUE, is promised to be one — one.
```

Axiomatic annotations of the package state type can be used to express algebraic relationships between the package subprograms by methods similar to those advocated for the definition of

abstract data types (see e.g. [4, 6]). This is usually achieved by axioms relating different successor states.

Example: An algebraic specification of a symbol tab/e package:

```
package SYMTAB is

    OVERFLOW, NOT-FOUND : exception;

--: function SIZE return INTEGER range 0 . . N:
--: function "=" (SS, TT : SYMTAB'TYPE) return BOOLEAN:
    -- Anna permits redefinition of " = " on limited types.

    function IN_BLOCK( S : STRING) return BOOLEAN:

    procedure INSERT(S : STRING: I : TOKEN):
--|     where raise OVERFLOW => in SYMTAB.SIZE = N:

    function LOOKUP( S : STRING) return TOKEN:
--|     where raise NOT-FOUND ;

    procedure ENTERBLOCK;

    procedure LEAVEBLOCK;

--| axiom
--| for all SS : SYMTAB'TYPE; S, T : STRING; I : TOKEN =>
--| SYMTAB'INITIAL[LEAVEBLOCK]     = SYMTAB'INITIAL,
--| SYMTAB'INITIAL.IN_BLOCK(S)     = FALSE,
--| SS[ENTERBLOCK; LEAVEBLOCK]     = ss,
--| SS[ENTERBLOCK].IN_BLOCK(S)     = FALSE,
--| SS[ENTERBLOCK].LOOKUP(S)       = SS.LOOKUP(S),
--| SS[INSERT(S, I); LEAVEBLOCK] = SS[LEAVEBLOCK],
--| SS[INSERT(S, I)].IN_BLOCK(T) =
--|     if S = T then TRUE else SS.DEFINED(T) end if,
--| SS[INSERT(S, I)].LOOKUP(T) =
--|     if S = T then I else SS.LOOKUP( T) end if;

end SYMTAB ;
-- The axioms define the semantics of the SYMTAB package algebraically by relating
-- successor states resulting from sequences of package operations. The virtual equality
-- operator, " = ", is predefined in Anna for package state types, but may be redefined in
-- the package body.
```

As illustrated in the SYMTAB example, visible annotations are intended to explain the behavior of a package to an outside user. This explanation must be independent of the implementation in the package body, simply as a result of the Ada visibility rules. However, additional properties of implementations may be introduced into the annotations — e.g., by virtual subprogram declarations — that would not be obvious from the actual Ada visible part. It should also be noted that explanation by means of formal annotations need not be “complete”. For example, the SYMTAB specification does not give the user any information about the state of the package after a NOT-FOUND exception has been propagated.

Package axioms also provide a means of specifying the properties of the underlying domain of values of a program. Thus the background knowledge that forms the basis for the construction of a program and its correct functioning can be "packaged" into descriptive theories. For example, an axiomatic theory of sorting could be given by means of a virtual package as follows:

```
--: generic
--:     type ITEM is private:
--:     type ENUM is (<>);
--:     type VECTOR is array (ENUM) of ITEM:
--: package SORTING-THEORY is
--:     function PERMUTATION(A, B : VECTOR) return BOOLEAN:
--:     function ORDERED( A : VECTOR) return BOOLEAN;
        ... -- Axioms defining mathematical properties of PERMUTATION and ORDERED.
--: end SORTING-THEORY;
```

The concepts, PERMUTATION and ORDERED can be used to specify actual sorting packages. It might be noted that such descriptive theories may also be executable if their packages have bodies; applications of executable specifications are still to be researched.

5. Semantics and Implementation of Anna

The meaning of each kind of annotation is described in terms of properties that must be satisfied by computations of the Ada text — or, in the case of a context annotation, by the Ada text itself — in its scope (Anna84 [2]). If the computations satisfy all these properties, then the Ada text is *consistent* with the annotations. This informal description of consistency suffices as a definition of the semantics of Anna in the same way as the Ada manual [1] provides a definition of Ada. The Anna programmer should be able to use annotations to explain Ada programs on the basis of the Anna manual.

In addition to the informal semantics, formal definitions of the meaning of Anna are needed to provide a basis for automating various possible applications of Anna. Two formal definitions of Anna (or more accurately, most of Anna) have been developed: an axiomatic definition, and an operational definition in terms of transformations of annotations into Ada text.

5.1 Axiomatic Semantics of Anna

The axiomatic definition of Anna semantics defines the atomic proof steps in mathematical proofs of consistency between Ada text and annotations. Ir can be used as a basis for constructing program verifiers for Anna. These are systems which mechanize consistency proofs, and in more sophisticated systems will aid in locating errors if consistency cannot be proved (see [15], also [17]). Such verification systems are potentially very useful in analyzing programs (or in the more general case, program designs in which units have been specified prior to implementation). However, experience has shown that production quality verifiers are substantial undertakings. They depend on subsidiary systerns, in particular for support of mechanized reasoning, that still have not achieved the necessary capabilities and power. We would therefore expect that verifiers for Ada that are based on Anna or subsets of Anna will take several years to develop.

5.2 Transformational Semantics and Runtime Checking

An alternative and complementary definition of Anna semantics is given in form of transformations on annotations (see [2, 9]). Here, the meaning of a more powerful annotation is defined in terms of sets of simpler annotations that are equivalent to it; for instance, the meaning of a type annotation is defined by transforming it into object constraints and subprogram annotations. In this way, most annotations are reduced to the simplest kind, *assertions.* An assertion is finally translated into Ada text that checks whether the assertion is satisfied by a program state. This checking code can then be executed together with the underlying Ada program, thus performing runtime error checking. The transformations can be implemented as a preprocessor to a standard Ada compiler. The resulting system enables Ada programs to be tested for consistency with their formal specifications; Anna exceptions will be raised automatically at points of inconsistency.

5.3 Anna Tools

The transformational semantics of Anna may serve as the basis for a variety of Anna tools. Debugging tools based on the transformational semantics can be developed quite quickly. Furthermore, crucial specifications (e.g., security requirements in databases) will be transformed automatically into correct runtime checks and may remain permanently in the final production version of a system. Here the optimization of runtime consistency checks presents a challenging problem, but this already is an issue in construction of Ada compilers. Smart static analysis will make compilation of many of the runtime checks unnecessary.

Finally, the syntactic and semantic analysis of Anna programs affords opportunity to catch many errors that may be missed by a standard Ada compiler simply because it ignores the annotations. For example, checking of propagation annotations and context annotations will catch unspecified use of exceptions and global variables. This kind of error checking can be applied to incomplete (or prototype) specifications of programs in Anna. It can be useful at very early stages in program development and is easy to implement. It may be expected that future versions of Anna will provide more kinds of declarative annotations (similar to context annotations) for expressing dependency and dataflow.

References

[1] *The Ada Programming Language Reference Manual*
US Department of Defense, US Government Printing Office, 1983.
ANSI/MILSTD 1815A Document.

[2] Luckham D.C., von Henke, F.W., Krieg-Brueckner, B., and Owe, 0.
ANNA: A Language for Annotating Ada Programs.
Computer Systems Laboratory Technical Report 84-261, Stanford University, July, 1984.
Program Analysis and Verification Group Report No. 24.

[3] Dahl, O.J.
Can Program Proving be Made Practical?.
Institute of Informatics, University of Oslo, May, 1978.

[4] Guttag, J.V.
Abstract Data Types and the Development of Data Structures.
Communications of *the ACM* 6(20):396-404, June, 1977.

[5] Hoare, C.A.R.
An Axiomatic Basis for Computer Programming.
Communications of the ACM 12(10):576-581, October, 1969.

[6] Hoare, C.A.R.
Proof of Correctness of Data Representations.
Acta Informatica (4):271-281, 1972.

[7] Hoare, C.A.R.
Monitors: An Operating System Structuring Concept.
Communications of the ACM 17(10):549-557, 1974.

[8] Hoare, C.A.R. and Wirth, N.
An Axiomatic Definition of the Programming Language Pascal.
Acta Informatica 2:335-355, *1973.*

[9] Krieg-Brueckner, B.
Consistency Checking in Ada and Anna: A Transformational Approach.
Ada Letters III(2):46-54, September,October, 1983.

[10] Luckham, D.C. and Polak, W.
A Practical Method of Documenting and Verifying Ada Programs with Packages.
In *Proceedings of the Symposium on the Ada Programming Language.* ACM, November, 1980.
ACM SIGPLAN Notices 15(11): 113-122.

[11] Luckham, D.C. and Polak, W.
Ada Exception Handling: An Axiomatic Approach.
ACM Transactions on Programming Languages and Systems 2(2):225-233, April, 1980.

[12] Luckham, D.C., and Polak, W.
Ada Exceptions: Specification and Proof Techniques.
CSD Report Prograrn Verification Group Report PVG- 16, STAN-E-80-789, Stanford University, February, 1980.

[13] Luckham, D.C. and Suzuki, N.
Verification of Array, Record and Pointer Operations in Pascal.
ACM Transacations on Programming Languages and Systems 1(2):226-244, October, 1979.

[14] Parnas, D., and Bartussek, W.
Using Traces to Write Abstract Specifications for Software Packages.
Technical Report UNC Report TR77-012, University of North Carolina, December, 1977.

[15] Luckham, D.C., German, S.M., vonHenke, F.W., Karp, R.A., Milne, P.W., Oppen, D.C., Polak, W., and Scherlis, W.L.
Stanford Pascal Verifier User Manual.
Technical Report Program Verification Report PV-11, CSD Report STAN-CS-79-731, Stanford University, March, 1979.

[16] *Process Design Language/Ada* .
IBM, 6600 Rockledge Drive, Bethesda, Maryland 20817, 1983.
Federal Systems Division, Ada Coordinating Group.

[17] Polak, W.
Lecture Notes in Computer Science. Volume 124: *Compiler Specification and Verification.*
Springer-Verlag; New York, 1981.