The Complete Transformation Methodology for Sequential Runtime Checking of An Anna Subset

Sriram Sankar
David Rosenblum

Technical Report No. CSL-M-301

Program Analysis and Verification Group Report No. 30

June 1988

The work was supported by the Advanced Research Projects Agency, Department of Defense, under Contract N00039-84-C-0211 and by Nasa Grant NAGW-419.

The Complete Transformation Methodology for Sequential Runtime Checking of An Anna Subset

CSL T.R. 86-301

Sriram Sankar
David Rosenblum

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

Abstract

We present in this report a complete description of a methodology for transformation of Anna (Annotated Ada) programs to executable self-checking Ada programs. The methodology covers a subset of Anna which allows annotation of scalar types and objects. The allowed annotations include subtype annotations, subprogram annotations, result annotations, object annotations, out annotations and statement annotations. Except for package state expressions and quantified expressions, the full expression language of Anna is allowed in the subset. The transformation of annotations to executable checking functions is thoroughly illustrated through informal textual description, universal checking function templates and several transformation examples. We also describe the transformer and related software tools used to transform Anna programs. In conclusion, we describe validation of the transformer and some methods of making the transformation and runtime checking processes more efficient.

Keywords: Ada, Anna, annotation language, checking function, checking semantics, constraint checking, formal specification, runtime checking, self-checking programs, transformation methodology, validation.

Computer Systems Laboratory
Stanford University

Table of Contents

1. INTRODUCTION

Anna [2, 5] is a language extension of Ada' [1] to include facilities for formally specifying the intended behavior of Ada programs. The extensions fall into three categories: (1) generalizations of Ada constraints, (2) addition of new kinds of constructs, mostly declarative in nature, and (3) addition of new specification constructs based on previous studies of the theory of program specification. These constructs appear as formal comments within the Ada source text. From the point of view of Ada, formal comments are just comments; in an Anna program, however, the formal comments must obey the syntactic and semantic rules of Anna. Anna defines two kinds of formal comments, which are introduced by special comment indicators in order to distinguish them from informal comments. These formal comments are *Virtual Ada* text, each line of which starts with the indicator "--:", and *Annotations,* each line of which starts with the indicator "--|".

The purpose of virtual text is to define concepts used in annotations. Often the specification of a program will refer to concepts that are not explicitly implemented as part of the program. These concepts are defined by virtual Ada declarations and bodies. Virtual text may also be used to compute values that are not computed by the actual program, but that are useful in defining the behavior of the program. Virtual text must be legal Ada. First of all, 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). Secondly, it must not influence the computation of the underlying Ada program by changing the values of actual objects. Thirdly, virtual declarations may not hide entities declared in the underlying program.

Annotations are built up from BOOLEAN-valued expressions and reserved words that indicate their kind and meaning. Anna provides different kinds of annotations, each associated with a particular Ada construct. 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. 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. Expressions that are permitted to occur in annotations include normal Ada expressions as well as quantified expressions, conditional expressions and Anna membership tests. .

---

[1] Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.

Some Anna examples are shown in Appendix I.

Anna is suited for a variety of applications. Some applications are:

- **Formal Documentation:** Anna specifications provide a formal notation for documenting the behavior of an Ada program for humans. The annotations complement the behavioral description provided by the Ada text.
- **Rapid Prototyping:** Anna can be used to provide a quick set of specifications for a particular problem. These specifications would be at a higher level and much more reliable than the actual Ada code that will eventually be written. These specifications can be implemented (generally) more inefficiently than the final code and they can be used to determine the correctness of the final code.
- **Testing and Debugging:** A *transformation* tool can be used to compile the Anna program (including the Anna specifications). The compiled code can be made to interact with other tools (e.g., a symbolic debugger). When the transformed program is executed, inconsistencies between the Anna specifications and the underlying Ada program are automatically detected and reported.
- **Production Quality Programs:** The Anna runtime checks could be left permanently in the program. A transformation tool with good optimizing capabilities could produce a program that is self-checking and that would report the occurrence of any inconsistencies during program operation, therefore creating an opportunity for automatic error recovery.

The semantics of Anna have been defined using two different approaches-Axiomatic Semantics and Checking Semantics. Axiomatic Semantics provide a basis for a mathematical proof of consistency between the formal specifications written in Anna and the underlying Ada program [4]. Checking Semantics is a set of transformations that convert Anna annotations into executable Ada text. This resulting Ada text checks *automatically* at runtime the consistency of the underlying (original) Ada program against the (original) annotations. Inconsistencies are reported by the propagation of various exceptions together with diagnostic information and possible interaction with a symbolic debugger. Section 9.2 describes the Anna Debugger and how the transformation system can be used to aid in debugging.

This paper describes in detail an implementation of the Checking Semantics for a subset of Anna. It is intended to extend the implemented subset to include other, more complex, features of Anna. Section 2 defines the already implemented subset. Section 3 gives an overview of the transformation methodology and describes the environment in which it runs. Section 4 describes the concept of a Checking Function and Sections 5,6,7 and 8 discuss the various steps in the transformation process with a lot of illustrative examples. Section 9 concludes with a description of the current status and

proposed future work. Section 9 also describes the Anna validation suite being developed. For an overview of the implementation methodology, please refer to [8] and [9].

2. DEFINITION OF THE SUBSET

The currently implemented subset of Anna is restricted to annotations of the Pascal-like constructs of Ada. Types and objects that can be annotated have been restricted to scalars only. Object annotations, subtype annotations, statement annotations, and subprogram annotations are included in the subset. Special Anna attributes have been excluded, and the predefined Anna attribute DEFINED is always assumed to be TRUE. All Anna expressions are permitted with the exception of quantified expressions and states. Section 5 describes in detail all Anna expressions that are in the subset.

The subset requires all virtual units to be provided with bodies. The identifier STANDARD can only be used to refer to the predefined package STANDARD. This restriction guarantees that in all cases, full names can be used to refer to objects even when they are hidden by other declarations. The following example illustrates the need for this restriction.

```
procedure P is
    A:INTEGER;
    procedure STANDARD is
        A,P:FLOAT;
    begin
        - - there is no way to refer to the INTEGER
        -- variable A at this point.

    end' STANDARD;
begin
    . .
end' P;
```

Tasking is permitted, so long as care is taken to ensure that the execution of a task does not cause annotations on any other task to be violated*. This is, however, not checked for by the current implementation.

For the syntax summary of the currently implemented subset, please refer to Appendix II.

---

[2] This is the *tasking assumption* that is assumed throughout this report (also see section 8.1)

3. AN OVERVIEW OF THETRANSFORMATION METHODOLOGY

The transformation methodology has been used to implement a *Transformer* that transforms the parse tree of the Anna program to the parse tree of an equivalent Ada program [6]. The equivalent Ada program is the underlying Ada text and virtual text (with the formal comment symbols removed) of the Anna program together with executable checks inserted at appropriate places to check annotations. The Anna parse tree is an extended form of a DIANA tree [3], while the Ada parse tree is a regular DIANA tree. The Transformer executes in an environment that contains the following utilities:

AST package: The AST (Abstract Syntax Tree) package is an implementation of DIANA. This package contains the data abstraction which defines the enhanced DIANA trees as well as the operations that can be performed on these trees.

Parser: A Parser is required to parse the input Anna programs and generate trees using the AST package.

Symbol table package:
A symbol table package with the capability to store various Anna entities, such as annotations, is required. This can be a regular Ada symbol table package enhanced for the purpose of transforming Anna.

Pretty-printer (unparser):
This transforms a tree defined by the above AST package to an Ada text file.

Overload-resolution package:
This is required to resolve ambiguities in annotations and expressions before various transformations can be applied to them. This package must use the previously mentioned symbol table package.

Semantic Processor:
Before the actual transformation of Anna programs, the semantic processor is used to check for the semantic correctness of these Anna programs. Among other things, it checks for the correctness of virtual text, conformance of annotations, and ensuring that annotations do not read or modify global variables when they are evaluated.

In the process of this transformation, a number of constructs, statements, etc. are introduced. The most important is the *Checking Function* which is described in detail in section 4. The transformation process involves, therefore, the introduction of many new identifiers. These identifiers must be such that they do not clash with identifiers in the input Anna program. For this purpose, the Transformer generates identifiers that start with the letter "O" followed by a six-digit serial number. It is assumed that the Anna source program does not contain such identifiers. However, for the purpose of this paper, the use of such identifiers can be very confusing, and so readable identifiers are used. These

Transformer-generated identifiers will appear *italicized* in this paper to distinguish them from other identifiers.

The transformation takes place in three logical steps. The first step is to make a few preliminary transformations. The most important of the preliminary transformations is the transformation of Anna expressions to Ada expressions. This step is described in detail in section 5. There are many other minor preliminary transformations which are described in section 6. The second step is to transform annotations into more basic annotations. *Subtype annotations, object constraints, result annotations* and *out annotations* are considered to be the basic annotations to which all other annotations are transformed; note that an *in annotation* is simply a trivial object constraint. These transformations are described in section 7. The third step is to transform the basic annotations into checking code. This usually involves the introduction of checking functions and is described in section 8. Note that these three steps are interleaved. For example, the transformation of the Anna membership test **is in** requires the appropriate checking function to have already been generated.

There are some standard entities which are referred to in the transformed Ada program, such as the predefined Anna exception ANNA-ERROR Such entities are defined in an Ada library package, which is shown below:

```
package ANNA is

   ANNA-ERROR : exception ;
   -- The predefined Anna exception. This exception should
   -- really be-declared in package STANDARD; however,
   -- the transformations described in section 6.2 makes this
   -- declaration completely equivalent to a declaration in
   -- package STANDARD.

   ANNA-EXCEPTION: exception ;
   -- An auxiliary exception that is needed for correct
   -- implementation of checking functions.

   procedure REPORT-ERROR;
   -- procedure that reports Anna errors.

   function CHECK-EXP(X: in BOOLEAN) return BOOLEAN:
   -- tests for the truth/falsity of X. If false, then an
   -- exception is raised. If true, the function returns X.

   procedure CHECK-EXP(X:in BOOLEAN) ;
   -- tests for the truth/falsity of X. If false, then an
   -- exception is raised.

end ANNA:
```

All Anna compilation units are transformed to include the following context clause (also see section *6.2):*

with *ANNA:* **use** *ANNA:*

4. CHECKING FUNCTIONS

The concept of a Checking Function is used extensively in the transformation methodology. A checking function is an Ada function that checks for the validity of annotation(s) and takes an appropriate action. This action could either be to return a BOOLEAN value that specifies the result of the check, or to raise an exception if the check shows that the annotation was violated. At various places where it is possible that annotations can be violated, appropriate checking functions are called. The advantages of implementing checks as calls to checking functions instead of expanding the checks inline are quite apparent. It is much simpler to generate checking functions and calls to them since the annotations do not have to be stored for too long by the transformation program; only the names of the generated checking functions have to be stored. Since it is usually the case that the same annotation has to be checked for validity at more than one place, the checking function approach lowers space requirements. This approach also saves space requirements when checking functions call other checking functions. Anyway, the effect of expanding checks inline can still be achieved by using the Ada pragma INLINE.

Sometimes checking functions have to call other checking functions. This has to be done when there is a nesting of annotations. An example of nesting is when a subtype of another subtype is declared, and both of these subtypes are annotated. In this case, the checking function of the new subtype calls the checking function of the other subtype. This ensures that when this new checking function is called, both annotations are checked as required by the Anna semantics.

Checking functions are defined currently for three kinds of annotations. These are subtype annotations, object constraints, and result annotations. Checking functions that raise an exception (if the annotation is found to be false) are generated for each of the above kinds of annotations. These checking functions are used to check validity of annotations at places in the program where it is possible that they can be violated. The structure of these checking functions is shown below; the braces indicate zero or more occurrences of the enclosed entity.

```
function CHECKING_FUNCTION( X: T) return T is
   { declarations ; }
begin
   {if not ( annotation ) then
      raise ANNA-EXCEPTION;
   end if ; }
   return X;
   -- In case another checking function has to be called, then the above
   -- return statement would be replaced by:
   --    return THE-OTHER-CHECKING-FUNCTION{ X) ;
exception
   when ANNA-EXCEPTION =>
```

```
        REPORT_ERROR;
        -- a procedure in package ANNA that raises ANNA-ERROR.
    when ANNA-ERROR =>
        raise ;
        -- ANNA-ERROR has a/ready been raised. So it is made to
        -- propagate out as is.
    when others =>
        -- An Ada exception was raised during ?he evaluation of
        -- an expression.
        give-an-error-message ;
        raise ANNA-ERROR;
end CHECKING-FUNCTION;
```

For subtype annotations there is one additional checking function defined. This function returns **TRUE** if the annotation is satisfied, and it returns FALSE otherwise. This checking function is used **to** implement the Anna membership test is **in** (section 5.1). It is also used to check object declarations (section 8.1.1) and initial values of formal parameters (section 8.1.3). The structure of this checking function is shown below.

```
function IS_IN_CHECKING_FUNCTION( X : T) return BOOLEAN is
begin
    return annotation ;
    -- In case another checking function has to be called, then the above
    -- return statement would be replaced by:
    --      return THE_OTHER_IS_IN_CHECKING_FUNCTION( X) and annotation ;
exception
    when ANNA-ERROR =>
        raise ;
        -- ANNA-ERROR has already been raised. So it is made to
        -- propagate out as is.
    when others =>
        -- An Ada exception was raised during the evaluation of
        -- an expression.
        give-an-error-message ;
        raise ANNA-ERROR;
end IS_IN_CHECKING_FUNCTION;
```

All checking functions are generated at the beginning of the later declarative region of the declarative part where the annotation is declared. Furthur details of each kind of checking function and their use are described in section 8.

5. TRANSFORMATION Of- ANNA EXPRESSIONS TO ADA EXPRESSIONS

This section describes in detail the transformation of Anna expressions to Ada expressions. Some transformations involve introduction of new objects, and hence new declarations. It will be seen in section 7 that all Anna expressions occur in basic declarative regions or get moved into a basic declarative region as a result of a transformation. Therefore, all declarations that are introduced as a result of transforming Anna expressions to Ada expressions can be inserted immediately preceding the transformed expressions. Anna expressions are transformed into Ada expressions through a four stage process; each stage is described separately in the subsections that follow.

5.1. Transformation of Anna Membership Operators and Implication Operators

The implication operators -> and <-> are equivalent to the predefined Ada relational operators <= and = respectively. However, the implication operators have lower precedence than the relational operators. This is automatically taken care of since the Transformer generates a function call to the predefined operator, rather than an infix relational operation.

Example:

```
A < B  - >  B > A
```

is transformed to

```
STANDARD."<="(A < B , B > A)
```

The Anna membership test is **in** is equivalent to a combination of the Ada membership test **in** and possibly a call to the checking function described in section 4. The call to the checking function is generated only when a type-mark is used in the membership test and there exists a checking function for this type-mark.

Examples:

```
X is in INTEGER
```

is transformed to

```
X in INTEGER
```

```
X is in EVEN
```

is transformed to

(X in EVEN) and then (*IS_IN_CHECKING_FUNCTION_FOR_EVEN(X)*)

X is in MIN . . MAX

is transformed to

X in MIN . . MAX

Section 8.1 describes in more detail the checking functions used for the transformation of Anna membership tests.

5.2. Transformation of Conditional Expressions

Except for three special cases discussed below, the following two realizations of a canonical conditional expression

F(if C_1 **then** E_1 **elsif . . . else** E_n **end if)**

and

if C_1 **then** $F(E_1)$ **elsif . . . else** $F(E_n)$ **end if**

are equivalent, since annotations have no side effects.

For all examples in this section, please assume the following type and object declarations:

```
type RELIGION-TYPE = (ISLAM, JUDAISM, CHRISTIANITY);
type SABBATH-DAYS  = (FRIDAY, SATURDAY, SUNDAY):
type RELIGION-REC(ATHEIST:BOOLEAN := FALSE) is record
    case ATHEIST is
        when TRUE =>
            null;
        when FALSE =>
            RELIGION:RELIGION-TYPE;
            SABBATH:SABBATH_DAYS;
    end case;
end record ;

R: RELIGION-REC;
```

Example:

```
R. SABBATH  = if R.RELIGION = ISLAM then
                  FRIDAY
              elsif R.RELIGION = JUDAISM then
```

```
        SATURDAY
    else
        SUNDAY
    end if;
```

The above annotation is equivalent to the following annotation:

```
if R.RELIGION = ISLAM then
    R.SABBATH = FRIDAY
elsif R.RELIGION = JUDAISM then
    R.SABBATH = SATURDAY
else
    R.SABBATH = SUNDAY
end if;
```

The three special cases where the above expressions are not equivalent are now described.

1. F(*cond_exp*) is of the form **f(in(g(***cond_exp***)))**, i.e. *cond_exp* is within an initial expression. In this case the two expressions

F(if C_1 then E_1 elsif . . . else E_n end if)

and

if in(C_1) then F(E_1) elsif . . . else F(E_n) end if

are equivalent.

Example:

```
in (R.SABBATH = if R.RELIGION = ISLAM then
                    FRIDAY
                elsif R.RELIGION = JUDAISM then
                    SATURDAY
                else
                    SUNDAY
                end if
   );
```

is equivalent to:

```
if in (R.RELIGION = ISLAM) then
    in (R.SABBATH = FRIDAY)
elsif in (R.RELIGION = JUDAISM) then
    in (R.SABBATH = SATURDAY)
else
    in (R.SABBATH = SUNDAY)
end if;
```

2. F contains a short-circuit operation. For example, F(*cond_exp*) could be of the form f (g and then h(*cond_exp*)). In this case, if g is false, then the conditional expression should not be

evaluated, which is not the case in the second canonical conditional expression,

Example:

```
not R.ATHEIST and then R.SABBATH =
            if R.RELIGION = ISLAM then
                FRIDAY
            elsif R.RELIGION = JUDAISM then
                SATURDAY
            else
                SUNDAY
            end if;
```

is *not* equivalent to

```
if R.RELIGION = ISLAM then
    not R.ATHEIST and then R.SABBATH = FRIDAY
elsif R.RELIGION = JUDAISM then
    not R.ATHEIST and then R. SABBATH = SATURDAY
else
    not R.ATHEIST and then R.SABBATH = SUNDAY
end if:
```

In the first case, R. RELIGION = ISLAM will not be evaluated if **not** R. ATHEIST evaluates to FALSE, whereas in the second case, R. RELIGION = ISLAM is evaluated **always.**

3. F contains another conditional expression. For example, F(*cond_exp*) could be of the form f **(if c then** g(*cond_exp*) else h **end if).** In this case, if c is false, then the conditional expression should not be evaluated, which is not the case in the second canonical conditional expression.

Example:

```
if not R.ATHEIST then
    R. SABBATH = if R.RELIGION = ISLAM then
                     FRIDAY
                 elsif R.RELIGION = JUDAISM then
                     SATURDAY
                 else
                     SUNDAY
                 end if
else
    TRUE
end if;
```

is not equivalent to:

```
if R.RELIGION = ISLAM then
    if not R.ATHEIST then
        R.SABBATH = FRIDAY
    else
```

```
            TRUE
        end if
    elsif R. RELIGION = JUDAISM then
        if not R.ATHEIST then
            R.SABBATH = SATURDAY
        else
            TRUE
        end if
    else
        if not R.ATHEIST then
            R.SABBATH = SUNDAY
        else
            TRUE
        end if
    end if;
```

Here again, in the first case, R. RELIGION = ISLAM will not be evaluated if **not** R .ATHEIST evaluates to FALSE, whereas in the second case, R. RELIGION = ISLAM is evaluated always.

Putting together all these observations and the fact that the result type of all annotations has to be BOOLEAN[3], one can always find a BOOLEAN function F such that the following two expressions

F(if C_1 **then** E_1 **elsif . . . else** E_n **end if)**

and

if [in](C_1) **then** F(E_1) **elsif . . . else** F(E_n) **end if**

are equivalent. Note that in case 2, h is such a BOOLEAN function and in case 3, g is such **a** BOOLEAN function.

Examples:

For each of the examples given above, a function F that satisfies the above conditions, is now listed.

First example: **F(x) ≡ (R.SABBATH = x)**

Second example: **F(x) ≡ (in (R.SABBATH = x))**

Third example: **F(x) ≡ (R.SABBATH = x)**

Fourth example: **F(x) ≡ (R.SABBATH = x)**

---

[3] Result annotations do not have to have **BOOLEAN** expressions. However, after applying the transformations of section 6.6, these annotations also become **BOOLEAN.**

The first step in the transformation of conditional expressions is to transform the annotation such that the above condition holds for each conditional expression one by one. In order to increase the speed of the transformation, the parse tree of the annotation is searched bottom-up for conditional expressions. Though the actual process is somewhat complicated, what actually happens can be illustrated by considering just one conditional expression. Since the conditional expression has been transformed so that the above condition holds, this expression has the following form:

if C_1 **then** E_1 **elsif** C_2 **then** E_2 . . . **elsif** C_{n-1} **then** E_{n-1} **else** E_n **end if**

where all C_i's and E_i's are BOOLEAN.

This conditional expression can now be transformed into an Ada expression containing short-circuit operations as shown below:

```
(
        (C1                                                    and then E1 )
 or else (not C1 and then C2                                   and then E2 )
 .   .   .
 or else (not C1 and then not C2 . . . and then Cn-1           and then En-1 )
 or else (not C1 and then not C2 . . . and then not Cn-1       and then En )
)
```

Example:

```
if (R.RELIGION = ISLAM) then
     (R.SABBATH = FRIDAY)
elsif (R.RELIGION = JUDAISM) then
     ( R. SABBATH = SATURDAY)
else
     (R.SABBATH = SUNDAY)
end if;
```

is transformed to

```
(
        ((R.RELIGION = ISLAM) and then
             (R.SABBATH = FRIDAY))
 or else (not (R. RELIGION = ISLAM) and then
          (R.RELIGION = JUDAISM) and then
             (R.SABBATH = SATURDAY))
 or else (not (R. RELIGION = ISLAM) and then
          not (R.RELIGION = JUDAISM) and then
             (R.SABBATH = SUNDAY))
)
```

5.3. Transformation of Anna Relational Expressions

Anna relational expressions are of the form:

```
expression {relational-operator expression}
```

Example:

```
I <= J < K < N
```

The Anna relational expression is transformed into a conjunction of Ada relational operators. For example, the above Anna relational expression is transformed to the following Ada expression:

```
"<="(I,J) and "<"(J,K) and "<"(K,N)
```

5.4. Transformation of Initial Names and Initial Expressions

Initial names and expressions are transformed by introducing a constant declaration just before the annotation, and replacing the initial name or initial expression by that constant. The constant is initialized to the value of the initial name or expression. .

Examples (assume all variables to be of type INTEGER):

```
--| A = in A;
```

is transformed to

```
--: IN_A :constant INTEGER : = A;
--| A = IN-A;
```

```
--| out (C*C = in (A*A + in (B*B)));
```

is transformed to

```
--: NEW_EXPR:constant INTEGER := A*A + B*B;
--| out (C*C = NEW-EXPR);
```

A similar transformation is also done for subtype annotations since all program variables occurring in subtype annotations are evaluated during elaboration.

6. OTHER PRELIMINARY TRANSFORMATIONS

6.1. Removal of Virtual Text Indicators

The semantic processor has already checked that the Anna program is semantically correct. Therefore, the Transformer just converts all existing virtual text to actual Ada text by deleting all virtual text formal comment indicators.

6.2. Making the Anna Library Package Visible

As mentioned in section 3, all Anna compilation units are transformed to include the following context clause:

with *ANNA;* **use** *ANNA:*

This ensures visibility of all necessary Anna entities. The Anna reference manual specifies that the predefined Anna exception ANNA-ERROR is declared in package STANDARD. However, in this implementation, this exception is declared in the Anna library package. Therefore, all occurrences of STANDARD. ANNA-ERROR are transformed to *ANNA*.ANNA_ERROR.

6.3. Naming of Unnamed Blocks and Loops

There are many situations where fully qualified names are used to refer to various entities (example: calls to checking functions). For this purpose, all unnamed blocks and loops are given a name. Otherwise, entities within such blocks and loops cannot be referenced.

6.4. Introduction of a Local Copy for out Parameters

If an **out** parameter is used in an annotation, it would also have to be passed to the corresponding checking function. However, **out** parameters cannot be read in Ada. To solve this problem, a local copy of the **out** parameter is introduced as illustrated in the following example:

```
procedure P(X:out T) is
   .  .
begin.
   .  .  .
end P;
```

is transformed to

```
procedure P(X:out T) is
   LOCAL-COPY-OF-X:T:
   .  .  .
begin
   .  .  .
```

```
end P;
```

If T is an unconstrained array type, or a record type (can be a private record type) with discriminants, the declaration of the local copy is modified to include index constraints or discriminant constraints, the value of the constraints being the same as that of the out parameter.

Examples:

```
procedure P( X:out STRING) is
   LOCAL-COPY-OF-X:STRING(X'FIRST( 1) , . X'LAST( 1));
   . . .

procedure P( X:out RELIGION-REC) is
   LOCAL_COPY_OF_X:RELIGION_REC(ATHEIST => R.ATHEIST);
   . . .
```

All assignments to the **out** parameter (or to its components) are transformed to an assignment to the local copy (or to its components) followed by an assignment to the **out** parameter (or to its components) as shown below:

```
x := exp;
```

is transformed to

```
LOCAL-COPY-OF-X := exp ;
x := LOCAL-COPY_OF_X;
```

Now the local cop; can be used when checking annotations instead of the **out** parameter.

6.5. Transformation of exit Statements

exit statements of the form:

```
exit [ loop-name ] when condition:
```

are transformed to:

```
if condition then
   exit [ loop-name ];
end if;
```

This is done to facilitate checking of **out** annotations at **exit** statements. Such annotations should be checked only when the condition evaluates to TRUE. However this is not possible in the first of the above forms of exit statements.

6.6. Transformation of Result Annotations

For some of the transformations such as transformation of conditional expressions, an assumption is made that expressions in annotations are always of the type BOOLEAN. This however, is not true of result annotations of the form:

```
return expression;
```

Therefore, such result annotations are transformed to the following form:

```
return X:T => "="(X , expression):
```

where T is the result type of the annotated function.

The visibility of the equality operator at the position of the result annotation is ensured by having renaming declarations for equality operators as described in section 6.7.

6.7. Renaming Declarations

In most cases, the position of the annotation in the transformed Ada program (i.e. within the checking function) is different from its position in the corresponding Anna program. This could sometimes cause problems due to different visibilities at these two different places. Therefore, every entity appearing in an annotation is renamed using a new name, and the new name is substituted for the entity in the annotation. For example, the following annotation:

```
--| (X mod 2) = 0;
```

where X is of type INTEGER, is transformed to:

```
--: NEW-X: INTEGER renames X;
--: function NEW_MOD( X ,Y: INTEGER) return INTEGER renames "mod";
-- : function NEW_EQUAL( X ,Y: INTEGER) return BOOLEAN renames "=";

--| NEW_EQUAL(NEW_MOD(NEW_X,2),0);
```

For the transformation described in section 6.6, the equality operator for the appropriate type must be visible at the position of the result annotation. To ensure this, the equality operator is declared again by a renaming declaration after every subtype declaration of that type, if the equality operator has not already been declared in the same declarative region.

Example:

```
subtype EVEN is INTEGER;
subtype ODD is INTEGER:
```

is transformed to

```
subtype EVEN is INTEGER:
function "="(X,Y:EVEN) return EVEN renames STANDARD."=";
subtype ODD is INTEGER;
-- The equality operator is not declared again here since
-- it has already been declared in this same declarative
-- region immediate/y after the declaration of EVEN.
```

7. TRANSFORMATION OF ANNOTATIONS TO MORE BASIC ANNOTATIONS

7.1. Simple Statement Annotations

The scope of a simple statement annotation is the immediately preceding statement. If the simple statement annotation occurs at the beginning of a block, the scope of the annotation is an imaginary **null** statement just before the annotation. At the place of the annotation and the statement in its scope, a new block statement is inserted. The simple statement annotation is converted to an **out** annotation and placed in the declarative part of this block, while the statement in the scope of the annotation is placed in the statement part of this block.

Example:

```
declare
    A:INTEGER := 3;
begin
    --I A = 3;
    A := A + 1;
    --I A = in A + 1;
end;
```

is transformed to

```
declare
    A:INTEGER := 3;
begin
    declare
        --I out (A = 3);
    begin
        null;
    end;
    declare
        --I out (A = in A + 1);
    begin
        A := A + 1;
    end;
end;
```

7.2. Compound Statement Annotations

Compound statement annotations have a compound statement immediately following them, and this statement is the scope of the annotation. If the compound statement is not a block statement, **a** new block statement is inserted in the place of the original compound statement. The original compound statement is now placed in the statement part of this new block statement. The compound statement annotation is transformed to an object annotation (either object constraints or **out**

annotations) and placed in the declarative part of the block statement. If the compound statement is already a block statement, then the compound statement annotation is transformed to an object annotation and placed at the beginning of the declarative part of this block statement. No new block statement is created in this case.

Example:

```
--| with
--|      A mod 2 = 0;
for I in J . . K loop
    A := A + 2;
end loop:
```

is transformed to

```
declare
    --I A mod 2 = 0;
begin
    for I in J . . K loop
        A := A + 2;
    end loop;
end ;
```

7.3. Subprogram Annotations

Subprogram annotations are transformed to object annotations (either object constraints or **out** annotations) and placed at the beginning of the declarative part of the subprogram.

Example:

```
procedure P( X: in out EVEN)
--I where
--|      X mod 2 = 0;
is
begin
    x := x + 2;
end P;
```

is transformed to

```
procedure P( X: in out EVEN) is
    --I X mod 2 = 0;
begin
    x := x + 2;
end P;
```

8. TRANSFORMATION OF BASIC ANNOTATIONS TO CHECKING CODE

8.1. Subtype Annotations

Subtype annotations are further constraints on Ada subtypes[4]. Each Ada subtype can, however, have only one subtype annotation associated with it. This annotation may only occur immediately after the subtype declaration.

Each subtype (excluding the predefined types and subtypes) has two checking functions associated with it, regardless of whether or not it has a subtype annotation. These functions have an empty declarative part and at most one test for the truth of an annotation, since subtypes can have at most one annotation. Nearly all user-defined scalar subtypes are defined in terms of some other (possibly predefined) subtype. If this other subtype has checking functions associated with it, then they are called by the checking functions of the newly-defined subtype. This is because subtypes inherit all constraints and annotations from the subtype in terms of which they were defined. The structure of the two checking functions is shown below. The second function (as mentioned in section 4) is used in the transformation of Anna membership tests (section 5.1). In the following two checking functions, assume that the subtype with which they are associated is T, and that the subtype in terms of which T is defined is U.

```
function CHECKING_FUNCTION( X : T) return T is
begin
    if not -( subtype-annotation ) then
        raise ANNA-EXCEPTION ;
    end if;
    -- The above if statement would not be present if T had no subtype
    -- annotation associated with it.
    return X;
    -- In case U has checking functions associated with it, then the
    -- above return statement would be replaced by:
    --      return T( CHECKING_FUNCTION_FOR_U( U( X))) ;
exception
    when ANNA-EXCEPTION =>
        REPORT-ERROR ;
        -- a procedure in package ANNA that raises ANNA-ERROR.
    when ANNA-ERROR =>
        raise ;
        -- ANNA-ERROR has already been raised. So it is made to
        -- propagate out as is.
    when others =>
        -- An Ada exception was raised during the evaluation of
```

[4] In this section, the term subtype refers to both types and subtypes

```
        -- an expression.
        give-an-error-message ;
        raise ANNA-ERROR:
end CHECKING-FUNCTION;

function IS_IN_CHECKING_FUNCTION( X : T ) return BOOLEAN is
begin
    return subtype-annotation ;
    -- In case U has checking functions associated with it, then the
    -- above return statement would be replaced by:
    --      return IS_IN_CHECKING_FUNCTION_FOR_U( U( X) )
    --                                          and subtype-annotation ;
    --
    -- If T did not have a subtype annotation then the two cases
    -- above would look like the following:
    --      return TRUE ;
    -- or
    --      return IS_IN_CHECKING_FUNCTION_FOR_U( U( X) ) ;
exception
    when ANNA-ERROR =>
        raise ;
        -- ANNA-ERROR has a/ready been raised. So it is made to
        -- propagate out as is.
    when others =>
        -- An Ada exception was raised during the evaluation of
        -- an expression.
        give-an-error-message ;
        raise ANNA-ERROR;
end IS_IN_CHECKING_FUNCTION;
```

Note the type conversions when the checking function for U is called. These are needed because it is possible for T to be a derived type of U and hence T and U are of different types. However, if T and U are of the same type, then the type conversions are not needed, but no reasonable compiler would introduce any extra code for the redundant type conversions.

Example:

```
declare
    type EVEN is new INTEGER:
    -- | where X:EVEN =>
    --      X mod 2 = 0;

    subtype POS_EVEN is EVEN:
--  --| where X:POS_EVEN =>
    --      x > 0;
begin
    . . .
end;
```

is transformed to

```
declare
    type EVEN is new INTEGER;
    subtype POS-EVEN is EVEN;

    function EVEN-CHECKING-FUNCT/ON( X:EVEN) return EVEN is
    begin
        if not ( X mod 2 = 0) then
            raise ANNA-EXCEPTION;
        end if:
        return X;
    exception
        when ANNA-EXCEPTION =>
            REPORT-ERROR;
        when ANNA-ERROR =>
            raise ;
        when others =>
            give-an-error-message ;
            raise ANNA-ERROR;
    end EVEN-CHECKING-FUNCTION;

    function IS_IN_EVEN_CHECKING_FUNCTION( X : EVEN) return BOOLEAN is
    begin
        return (X mod 2 = 0);
    exception
        when ANNA-ERROR =>
            raise ;
        when others =>
            give-an-error-message ;
            raise ANNA-ERROR;
    end IS_IN_EVEN_CHECKING_FUNCTION;

    function POS_EVEN_CHECKING_FUNCTION( X : POS-EVEN) return POS-EVEN is
    begin
        if not (X > 0) then
            raise ANNA-EXCEPTION;
        end if;
        return POS_EVEN(EVEN_CHECKING_FUNCTION(EVEN(X)));
    exception
        when ANNA-EXCEPTION =>
            REPORT-ERROR;
        when ANNA-ERROR =>
            raise ;
        when others =>
            give-an-error-message ;
            raise ANNA-ERROR;
    end POS_EVEN_CHECKING_FUNCTION;

    function IS_IN_POS_EVEN_CHECKING_FUNCTION( X: POS-EVEN)
                                                    return BOOLEAN is
    begin
        return IS_IN_EVEN_CHECKING_FUNCTION( X) and (X > 0) ;
    exception
        when ANNA-ERROR =>
            raise ;
        when others =>
```

```
            give-an-error-message ;
            raise ANNA-ERROR ;
        end IS_IN_POS_EVEN_CHECKING_FUNCTION;

    begin
        . . .
    end;
```

Now that the checking functions of subtype annotations have been explained, the following sections describe how subtype annotations are checked using these checking functions.

The transformation of the Anna membership test has already been described in section 5.1. The other places where subtype annotations have to be checked are:"

- Object declarations — The initial value of objects must satisfy the annotations on their subtype.
- Assignment statements — The assignment expression must satisfy the annotations on the subtype of the target object.
- Entry into subprograms — All formal parameters of mode **in** and **in out** must satisfy the annotations on their subtype.
- Exit from subprograms — All actual parameters of mode **in out** and **out** must satisfy the annotations on their subtype. If the subprogram is a function, then the value returned must satisfy the annotations on the return subtype.
- Type conversions — The expression being converted to a new type must satisfy the annotations of the new type.
- Qualified expressions — The expression being qualified must satisfy the annotations of the qualifying type.

8.1 .1. Object declarations

Immediately after an object declaration, dummy declaration(s) are inserted that test for the consistency of the object(s) with the annotations on their subtype. These dummy declarations have initialization expressions which are calls to functions in the Anna library package. These functions make the necessary tests. This is the only way to perform any sort of test in a basic declarative region. It is possible that some of the necessary checking functions have not yet been elaborated. This happens if the subtype of the object(s) is also declared in the same declarative region. In this case, the checking function cannot be called, and the annotation needs to be explicitly tested for. This subtype may depend on another subtype. Again the checking function for this subtype may or may not be elaborated. In the general case, there is a set of annotations that need to be explicitly checked (corresponding to all the relevant subtypes that are declared in the same declarative region),

---

'Note that we are making the tasking assumption here (see section 2)

and then the remaining annotations (corresponding to all the relevant subtypes that are *not* declared in the same declarative region) are checked by calling the checking function of the least global type that is not in the same declarative region.

Example:

```
declare
    subtype PERFECT-CUBE is INTEGER;
    --| where X:PERFECT_CUBE =>
    --|     (INTEGER(CUBE_ROOT(FLOAT(X))) ** 3) = X;
begin
    declare
        subtype EVEN-PERFECT-CUBE is PERFECT-CUBE;
        --| where X: EVEN-PERFECT-CUBE =>
        --|     (X mod 2) = 0;

        subtype POS-EVEN-PERFECT-CUBE is EVEN-PERFECT-CUBE;
        --| where X:POS_EVEN_PERFECT_CUBE =>
        --|     X > 0;

        A:PERFECT_CUBE := 0;
        B:EVEN_PERFECT_CUBE := 0;
        C:POS_EVEN_PERFECT_CUBE := 0;
    begin
        . . .
    end ;
end ;
```

is transformed **to**

```
declare
    subtype PERFECT-CUBE is INTEGER;
    . . .
begin
    declare
        subtype EVEN-PERFECT-CUBE is PERFECT-CUBE ;
        subtype POS-EVEN-PERFECT-CUBE is EVEN-PERFECT-CUBE;

        A: PERFECT-CUBE := 0;
        A-CHECK: BOOLEAN := CHECK-EXP(
          IS_IN_PERFECT_CUBE_CHECKING_FUNCTION(A));

        B:EVEN_PERFECT_CUBE := 0;
        W-CHECK: BOOLEAN := CHECK-EXP( (B mod 2) = 0);
        B2_CHECK: BOOLEAN := CHECK-EXP(
          IS_IN_PERFECT_CUBE_CHECKING_FUNCTION(PERFECT-CUBE(B)));

        C:POS_EVEN_PERFECT_CUBE := 0;
        C1-CHECK: BOOLEAN := CHECK-EXP( C > 0) ;
        C2_CHECK: BOOLEAN := CHECK-EXP(
          (EVEN-PERFECT-CUBE(C) mod 2) = 0);
        C3_CHECK: BOOLEAN := CHECK-EXP(
          IS_IN_PERFECT_CUBE_CHECKING_FUNCTION( PERFECT-CUBE(C) ) );
```

```
    begin
        . . .
    end ;
end ;
```

8.1.2. Assignment statements

The checking function of the subtype of the object being assigned to is called just before the assignment is made.

Example:

```
declare
    A:POS_EVEN;
begin
    A := 20:
end ;
```

is transformed to

```
declare
    A:POS_EVEN;
begin
    A : = POS_EVEN_CHECKING_FUNCTION( 20) ;
end ;
```

8.1.3. Entry into subprograms

Dummy declarations are introduced for **in** and **in out** parameters in a manner similar to that used for object declarations (section 8.1 .1). However, in this case, all relevant subtypes will be global, and so explicit checks for annotations are not required (only calls to checking functions are required).

Example:

```
procedure ABSOLUTE-VALUE(X:in EVEN;Y:inout POS-EVEN) is
begin
    . . .
end ABSOLUTE-VALUE;
```

is transformed to

```
procedure ABSOLUTE_VALUE( X : in EVEN ; Y: in out POS-EVEN) is
    X_CHECK:EVEN : = CHECK-EXP( IS_IN_EVEN_CHECKING_FUNCTION( X) ) ;
    Y-CHECK: POS-EVEN : =
        CHECK-EXP( IS_IN_POS_EVEN_CHECKING_FUNCTION( Y) ) ;
begin
    . . .
end ABSOLUTE-VALUE;
```

8.1 .4. Exit from subprograms

immediately after exit from subprograms (immediately following the subprogram call statement), dummy assignments are performed on all **in out** and **out** parameters. These dummy assignments call the appropriate checking functions. The reason why dummy assignments are preferred to checking procedures is explained in section 8.2. If the subprogram is a function, then a call to the checking function is made at all return statements.

Example:

```
function ABSOLUTE-VALUE (X : EVEN) return POS-EVEN is
    TMP:POS_EVEN;
begin
    ABSOLUTE-VALUE(X,TMP);
    -- see example of section 8.7.3
    return TMP;
end ABSOLUTE-VALUE;
```

is transformed to

```
function ABSOLUTE-VALUE( X: EVEN) return POS-EVEN is
    TMP:POS_EVEN;
begin
    ABSOLUTE-VALUE(X,TMP);
    TMP := POS_EVEN_CHECKING_FUNCTION( TMP);
    return POS_EVEN_CHECKING_FUNCTION( TMP) ;
end ABSOLUTE-VALUE;
```

8.1.5. Type conversions

Immediately after a type conversion, a call to the appropriate checking function is made. This is not done when the type conversion is an actual parameter of mode **in out** or **out;** otherwise, the resulting Ada program becomes illegal. However, these checks are made at other places (sections 8.1.3 **and** 8.1.4).

Example:

```
declare
    A:INTEGER;
    B: EVEN;
begin
    B := EVEN(A);
end ;
```

is transformed to

```
declare
    A: INTEGER;
    B:EVEN;
```

```
begin
    B := EVEN_CHECKING_FUNCTION(EVEN(A));
end;
```

8.1.6. Qualified expressions

A call is made to the appropriate checking function with the qualified expression as a parameter.

Example:

```
declare
    A,B:EVEN;
begin
    A := B + EVEN'(2);
end;
```

is transformed to

```
declare
    A,B:EVEN;
begin
    A := B + EVEN-CHECKING-FUNCTION (EVEN'(2));
end;
```

8.2. Object Constraints

Object constraints are constraints on one or more objects over a particular scope. Object constraints can occur in any basic declarative region, as subprogram annotations or as compound statement annotations. Subprogram annotations and compound statement annotations are, however, transformed to declarative annotations (section 7.2 and 7.3). An object can have any number of object constraints in different scopes constraining its value.

An object is a *constituent of an object constraint* if the object constraint constrains the object. To generate checking functions, all object constraints within the same declarative region are grouped together. Every object that is a constituent of at least one object constraint in this group has a checking function in this declarative region; i.e. these checking functions are associated with objects. The checking function has one check corresponding to each object constraint of which the object is a constituent. To simplify matters, each check in the checking function has associated with it a renaming declaration. This declaration associates the formal parameter of the checking function with the appropriate object. This is illustrated in examples later. In case an object has associated with it

more than one checking function (in different scopes), the more deeply nested checking functions make calls to the next most deeply nested checking function and so on until the most global checking function is reached. Thus, by making a call to a checking function associated with an object, all object constraints on that object are tested. The structure of this checking function is shown below.

```
function CHECKING_FUNCTION( X : T) return T is
    { renaming-declarations ; }
begin
    {if not ( object-constraint ) then
        raise ANNA-EXCEPTION;
    end if ; }
    return X;
    -- In case another checking function has to be called, then the above
    -- retu rn statement would be replaced by :
    --      return THE-OTHER-CHECKING-FUNCTION(X) ;
exception
    when ANNA-EXCEPTION =>
        REPORT-ERROR ;
        -- a procedure in package ANNA that raises ANNA-ERROR.
    when ANNA-ERROR =>
        raise ;
        -- ANNA-ERROR has a/ready been raised. So it is made to
        -- propagate out as is.
    when others =>
        -- An Ada exception was raised during the evaluation of
        -- an expression.
        give-an-error-message ;
        raise ANNA-ERROR;
end CHECKING-FUNCTION;
```

Example:

```
declare
    A,B:INTEGER := 1;
    --I A + B > 0;
    --I B > 0;
begin
    . . .
end;
```

is transformed to

```
declare
    A,B:INTEGER := 1;
    A1: INTEGER renames A;
    B1: INTEGER renames B;
    B2: INTEGER renames B;

    function CHECKING_FUNCTION_FOR_A( X: INTEGER) return INTEGER is
     A1:INTEGER renames X;
    begin
        if not (A1 + B1 > 0) then
```

```
            raise ANNA-EXCEPTION;
        end if;
        return X;
    exception
        when ANNA-EXCEPTION =>
            REPORT-ERROR;
        when ANNA-ERROR =>
            raise ;
        when others =>
            give-an-error-message ;
            raise ANNA-ERROR;
    end CHECKING_FUNCTION_FOR_A ;

    function CHECKING_FUNCTION_FOR_B( X: INTEGER) return INTEGER is
        B1: INTEGER renames X;
        B2: INTEGER renames X;
    begin
        if not (A, + B1 > 0) then
            raise ANNA-EXCEPTION;
        end if:
        if not (B2 > 0) then
            raise ANNA-EXCEPTION;
        end if;
        return X;
    exception
        when ANNA-EXCEPTION =>
            REPORT-ERROR ;
        when ANNA-ERROR =>
            raise ;
        when others =>
            give-an-error-message ;
            raise ANNA-ERROR;
    end CHECKING-FUNCTION-FOR-B ;

begin
    . . .
end ;
```

Note the renaming declarations immediately after the declaration of A and B. This has already been discussed in section 6.7. (Actually, the Transformer will rename "+" and ">" also.) Also notice how the renaming declarations within the checking functions simplify the generation of the checks (the annotations can remain as they are; it is not necessary for the formal parameter to be substituted into the annotations). There are some redundant renaming declarations; however this will not affect the speed of the transformed Ada program.

The checking functions that are generated for object constraints are called at the following places:

- Assignment statements — The expression being assigned must satisfy the object constraints on the target variable.
- Exit from subprograms — All actual parameters of mode **in out** and **out** must satisfy their

object constraints,

In both the above mentioned cases, the transformation is performed in exactly the same manner as is done in the corresponding cases for subtype annotations. In the second case (i.e. exit from subprograms), a dummy assignment of a checking function is preferred to using a checking procedure, otherwise, an extra checking function or checking procedure needs to be generated for each object.

Examples:

```
A := B + 1;
```

is transformed to

```
A := CHECKING_FUNCTION_FOR_A (B + 1);
```

```
TEXT_IO.GET(A);
```

is transformed to

```
TEXT-IO.GET(A);
A := CHECKING_FUNCTION_FOR_A (A);
```

In addition, object constraints (and in annotations) have to be tested at the point of occurrence of the annotation. This is done by inserting a dummy declaration that makes a call to a function in the Anna library package at this point. The function call does the necessary testing. In the first example of this section, there is one more transformation performed to make this test. The resulting program is shown below:

```
declare
    A,B:INTEGER := 1;
    A1: INTEGER renames A;
    B1: INTEGER renames B;
    B2: INTEGER renames B;

    CHECK_ANNO_1:BOOLEAN := CHECK-EXP(A, + B1 > 0);
    CHECK_ANNO_2: BOOLEAN := CHECK-EXP( B2 > 0);

    function CHECKING_FUNCTION_FOR_A( X: INTEGER) return INTEGER is
        . . .
    end CHECKING-FUNCTION-FOR-A ;

    function CHECKING_FUNCTION_FOR_B( X: INTEGER) return INTEGER is
        . . .
    end CHECKING_FUNCTION_FOR_B ;
```

```
begin
    . . .
end ;
```

8.3. Result Annotations

Result annotations are constraints on the values returned by functions. Result annotations can occur as declarative annotations, subprogram annotations or statement annotations. Subprogram annotations and statement annotations are, however, transformed to declarative annotations (section 7). Result annotations can either have or not have a logical variable. In case it does not, it is transformed to an equivalent form that has a logical variable (section 6.6). Each function can have any number of result annotations in different scopes constraining its return value.

Checking functions of result annotations are similar to checking functions of object constraints **A** function will have one checking function for each declarative region that contains result annotations. The checking function will contain one check for each result annotation in the declarative region. Just as in the case of object constraints, checking functions of result annotations also contain one renaming declaration corresponding to each annotation to simplify the generation of checks. In case a function has associated with it more than one checking function (in different scopes), the more deeply nested checking functions make calls to the next most deeply nested checking function and so on until the most global checking function is reached. Thus, by making a call to a checking function associated with a function, all result annotations on that function are tested. The structure of this checking function is shown below.

```
function CHECKING-FUNCTION{ X: T) return T is
    { renaming-declarations ; }
begin
    (if not ( result-annotation ) then
        raise ANNA-EXCEPTION;
    end if ; }
    return X;
    - - In case another checking function has to be called, then the above
    -- return statement would be replaced by:
    - -       return THE_OTHER_CHECKING_FUNCTION( X) ;
exception
    when ANNA-EXCEPTION =>
        REPORT-ERROR ;
```

which these annotations constrain.

Example:

```
return LN(F)/LN(10.0);
```

is transformed to

```
return CHECKING_FUNCTION_FOR_LOG( LN( F)/LN( 10.0)) ;
```

8.4. out Annotations

out annotations are constraints on the normal (non-exceptional) exit states of their scopes. After the transformations described in section 7, all **out** annotations will occur as declarative annotations. Hence the scope of the annotations will be either a block statement or the body of a subprogram, package, task unit, or generic unit. **out** annotations are tested for just before leaving their scope (normally). There are four normal ways of leaving the scope of an **out** annotation. They are

- reaching the end of the sequence of statements
- executing a **return** statement
- executing an **exit** statement
- executing a **got0** statement

For each point of exit, a list of all the annotations to be tested is determined. For each of these annotations, a test is generated by calling the procedure *CHECK-EXP* in the Anna library package *ANNA.* These procedure calls are inserted at the point of exit so that the annotations are tested just before leaving the scope, but after all other computation within the scope has been completed. To determine the list of the **out** annotations to be tested, the point where control is transferred after exit is first determined. All **out** annotations whose scope does not include this point, but includes the point of exit are the annotations that form the necessary list.

Example:

The list of annotations to **be** tested is enumerated within braces at each exit point in the following program:

```
procedure FOO is
   --| out F1;
begin
   declare
      --| out F2;
   begin
      loop
         declare
            --| out F3;
         begin
            . . .
            exit;  {F3}
            . . .
            return ;  {F1, F2, F3}
            . . .
            goto L;  {F2, F3}
            . . .
         end;  {F3}
      end loop:
      . . .
      return ;  {F1, F2}
      . .
   end; . {F2}
   .
<<L>> .
   goto L; {}
   . . .
end FOO;  {F1}
```

Each kind of exit point is now dealt with separately.

8.4.1. Reaching the end of the sequence of statements

The list of annotations for this case will be all the out annotations declared in the corresponding declarative part. Tests for each annotation are generated after the last statement in the sequence of statements.

Example:

```
procedure EXCHANGE (X, Y: in out INTEGER) is
   --| out (X = in Y);
   --| out (Y = in X);
   T : INTEGER;
begin
   x := x + Y;
   Y := x - Y;
   x := X - Y ;
exception
   when CONSTRAINT-ERROR =>
      T. . X;
```

```
        x := Y;
        Y := T;
end EXCHANGE;
```

is transformed to

```
procedure EXCHANGE(X,Y:inout INTEGER) is
    IN-Y :constant INTEGER := Y;    -- see section 5.4
    IN_X:constant INTEGER := X;
    T:INTEGER;
begin
    X := x + Y;
    Y := X - Y ;
    CHECK-EXP( X = IN-Y) ;
    CHECK-EXP( Y = IN-X) ;
exception
    when CONSTRAINT-ERROR =>
        T := X;
        x := Y;
        Y := T;
        CHECK-EXP( X = IN-Y) ;
        CHECK-EXP( Y = IN-X) ;
end EXCHANGE;
```

8.4.2. Executing a return statement

The list of **out** annotations includes all currently visible **out** annotations declared within the subprogram whose execution is being completed.

Example:

```
procedure SORT( X, Y: in out INTEGER) is
    --| out (X >= Y);
begin
    if X >= Y then
        retu rn ;
    else
        EXCHANGE(X,Y);
    end if;
end SORT:
```

is transformed to

```
procedure SORT( X,Y: in out INTEGER) is
begin
    if X >= Y then
        CHECK-EXP(X >= Y);
        return ;
    else
        EXCHANGE(X,Y);
    end if:
    . .
end' SORT;
```

If the return statement corresponds to a function and the return expression needs to be evaluated, it is possible for this evaluation to have side-effects. Therefore, the **out** annotations must be tested only *after* evaluating the return expression. For this purpose, a function is generated that contains all the necessary tests, and this function is called as shown in the example below. This approach has to be taken since there is no other easy way to perform the checks correctly. Hence this function is really a work-around and does not qualify as a checking function in the same sense as the other checking functions.

Example:

```
function C( N, R: INTEGER) return INTEGER is
    NUMERATOR,FACT_R:INTEGER;
    --| out (FACT-R = FACTORIAL(R));
    -- FACTORIAL has a/ready been defined and is visible here.
begin
    . . .
    return NUMERATOR/FACT-R;
    -- out annotation has to be tested after evaluation of
    -- NUMERATOR/FACT-R.
end C;
```

is transformed to

```
function C( N, R: INTEGER) return INTEGER is
    NUMERATOR,FACT_R:INTEGER;
begin
    . . .
    declare
        function OUT_CHECK( X: INTEGER) return INTEGER is
        begin
            CHECK-EXP( FACT-R = FACTORIAL(R));
            return X:
        end OUT-CHECK;
    begin
        return OUT_CHECK( NUMERATOR/FACT-R):
    end ;
end C;
```

8.4.3. Executing an exit statement

exit statements will be in their unconditional form because of the transformations described in section 6.5. The list of **out** annotations associated with an **exit** statement will consist of those declared in any block whose scope is being exited as a result of exiting the loop corresponding to the **exit** statement.

Example:

```
loop
    declare
        I,J:INTEGER;
        --I out (I = J);
    begin
        . . .
        exit when I <= 0:
        . .
    end;
end loop;
```

is transformed to

```
loop
    declare
        I,J:INTEGER;
    begin
        . . .
        if I <= 0 then   -- see section 6.5
            CHECK-EXP( I = J);
            exit ;
        end if;
    end ;
end loop:
```

8.4.4. Executing a goto statement

This case is similar to the preceding cases. The only difference is that in the case of **goto** statements, control may remain in the same scope even after the execution of the **goto** statement. Hence, in such a situation, the list of **out** annotations to be checked will be empty.

9. CONCLUSIONS AND FUTURE WORK

9.1. Current Status of the Transformer-Validation and Results

Validation of the Anna Transformer is carried out by executing the Transformer on a suite of Anna test inputs. Each of these tests contains a single kind of annotation in a particular context (e.g., object annotations specifying a mutual constraint on two objects). The suite is intended to test the Transformer on every conceivable Anna construct that can occur in a program using the currently implemented subset of Anna. The test suite, which currently contains close to 100 individual test programs, is continually expanded with new tests as we improve our understanding of the subtleties of the Anna semantics.

Validation of the Anna Transformer is a three-step process performed on each test program: (1) A test program is input to the Transformer, (2) The resulting transformed program is compiled by a validated Ada compiler to test for syntactic and semanticcorrectness, and (3) The checking functions that are produced are executed in some cases and visually inspected in other cases to ensure that the transformations were performed correctly. Syntax and semantic errors in annotations are detected by the Anna Parser and by the Semantic Processor, respectively. The Transformer has succeeded every step of this testing procedure on the complete validation suite.

9.2. An User Interface to the Transformer

A current project is the development of the *Anna Debugger* which is an interface to the Transformer that will simplify the debugging of annotated programs. This interface consists of a major enhancement of the diagnostic information provided during execution of a transformed program, including the location within the original (untransformed) source text at which an annotation was violated as well as the source location of the violated annotation(s). In addition, the Transformer will be extended with the capability of inserting calls to a symbolic debugger at places in the transformed program where the exception ANNA-ERROR could be raised. This capability will allow the user to query during execution of a transformed program the state of the program that caused an Anna constraint to be violated. The Anna Debugger will also have its own user interface which will provide the user with the capability to name annotations and modify the effects of violating Anna constraints. These modifications will include completely ignoring the constraint violations or just reporting the constraint violations without raising any exception.

9.3. Optimization of the Transformations

Another project being undertaken is the optimization of the checking functions produced by the Transformer as well as optimization of the algorithms used by the Transformer. These optimizations fall into three broad classes: (1) Merging of checking functions, (2) Static checking of annotations and (3) Parallel checking of annotations.

9.3.1. Merging of checking functions

The simplest of these optimizations is the merging of checking functions during the transformation and runtime checking of annotated programs. For example, the checking function for an annotated subtype which is derived from other annotated subtypes can in certain cases be merged with the other checking functions into a single checking function. The proper merging of checking functions is a simple extension of some of the recognition capabilities already available in the current Transformer. In fact, some checking functions can be deleted altogether.

9.3.2. Static checking of annotations

Certain kinds of annotations lend themselves well to static (i.e., "transformation-time") verification. Statically verified annotations would be ignored during the regular transformation process. Although static verification theorems and techniques have been studied for several years, it is not clear how easy it will be to apply such ideas to the complete Ada and Anna semantic framework.

9.3.3. Parallel checking of annotations

A class of optimizations currently being designed is the checking of annotations in parallel with the original program [7]. The basic idea of such optimizations is that checking tasks running on different processors replace the checking functions of a sequential transformation. Expressions to be checked are sent to an appropriate checking task via a shared queue, and then the checking task performs the annotation checks at its leisure.

There are several problems with this approach to annotation checking. Unlike a pure Ada program, an Anna program would be allowed to proceed with its execution in this model even if it has violated some (Anna) constraint. Thus, in order to perform parallel checking it is necessary to relax some of the Anna semantic rules concerning when constraint checking is to be performed, when the predefined exception ANNA-ERROR is to be raised, and how user-defined exception handlers for ANNA-ERROR are to be transformed. Some would argue that this defeats the whole purpose of constraint specification; however, further execution of a program would have a "non-destructive" effect if an annotation were violated. On the other hand, if it turns out that a constraint was not violated, then the original program has suffered a negligible slowdown. In either case, the checking

task must eventually signal the original program that a constraint was violated, after which the sequential semantics would take over; that is, the checking task will eventually "catch up" with the original program and notify the program of an ANNA-ERROR.

Parallel checking of constraints would be of great use in such time-critical applications as real-time embedded software; however, a highly intelligent checking model is needed to ensure that the reduction of execution overhead due to sequential checking is not negated by the lack of robustness caused by delayed notification of constraint violations.

9.4. Extending the Transformer to Recognize Full Anna

Although the Anna Parser and Semantic Processor are able to recognize the complete syntax of Anna, viable transformations for certain Anna constructs have yet to be determined. Currently under study is the enlargement of the subset of Anna that the Transformer can recognize to include annotation of composite types and objects, annotation of access types and objects, array states, record states, collections, quantified expressions, package annotations, context annotations and propagation annotations. To appreciate the difficulty in efficiently transforming some of these features, consider an example involving quantified expressions. A reasonable annotation of a prime number subtype can be accomplished in the following manner:

```
type DOMAIN is new POSITIVE range 2 . . POSITIVE'LAST;
subtype PRIME is DOMAIN:
--| where X : PRIME => for all Y,Z : DOMAIN => (X /= Y * Z);
```

The limited intelligence of the current Transformer allows a single option for transforming this annotation, namely the formation of two nested loops on the ranges of Y and Z, respectively, with the inequality checked at every iteration. In this case such a transformation is obviously impractical. Aside from expecting the programmer to state such annotations more succinctly, any other method of transformation would require the Transformer to deduce (using a system of algebraic axioms) which ordered pairs (Y , Z) actually need to be checked before the checking function is produced. In the general case, much more powerful analysis techniques will be needed to derive automatically a tractable impelementation of such intensive checks; such techniques are certainly beyond the analysis capabilities of most traditional compiler components. Nevertheless, this simple example clearly demonstrates the need for further research in determining how to efficiently check constraints expressed in a language as powerful **Anna.**

10. ACKNOWLEDGEMENTS

The authors wish to thank D. Luckham for his inspiration and guidance in the Anna implementation project. We are also grateful to R. Neff for his contribution to the implementation project, to F.W. von Henke for his contribution in determining the correctness of the individual checking transformations and to G.O. Mendal for his many helpful comments on the manuscript.

Appendix I
Examples of Annotations and Virtual Text

Object Annotations

```
CIRCUMFERENCE,DIAMETER:FLOAT;
--| CIRCUMFERENCE = PI * DIAMETER:
```

Subtype Annotations

```
subtype EVEN is INTEGER:
--| where X:EVEN =>
--|       X mod 2 = 0;
```

Statement Annotations

```
I := 2;
--| with
--|       1 < I and I <= N;
-- Compound statement annotation.
while I <= N loop
    if A(I-1) > A(1) then
        EXCHANGE(A(I-1),A(I));
    end if:
    --I A(1) >= A(I-1);
    -- Simple statement annotation.
    I := I+1;
end loop ;
```

Subprogram Annotations

```
procedure EXCHANGE(X,Y:inout ELEM);
--I where
--|       out(X = in Y), out(Y = in X):
```

Propagation Annotations

```
procedure PUSH(STACK:inout STACK-TYPE;E:in ELEM);
-- | LENGTH =MAX-LENGTH  => raise OVERFLOW;
--| raise OVERFLOW => (STACK = in STACK):
```

Context Annotations

```
--I limited to A,B,C;
declare
    . . .
begin
    . . .
```

```
end ;
```

Summary

An example of a completely annotated package, including virtual text, subprogram annotations and package axioms.

```
generic
    type ITEM is private;
    STACK-SIZE:NATURAL;
package STACK is

    OVERFLOW,UNDERFLOW:exception;

    --: function LENGTH return NATURAL:

    procedure PUSH( X: in ITEM);
    --| where
    --|     out( LENGTH = in LENGTH + 1),
    --|     LENGTH = STACK-SIZE => raise OVERFLOW,
    --|     raise OVERFLOW => (STACK = in STACK);

    function POP return ITEM;
    --| where
    --|     out( LENGTH = in LENGTH - 1),
    --|     LENGTH = 0 => raise UNDERFLOW,
    --|     raise UNDERFLOW => (STACK = in STACK);

--| axiom
--|     for all S:STACK'TYPE;X:ITEM =>
--|       ( STACK'INITIAL.LENGTH = 0,
--|         S[PUSH(X)].POP = X,
--|         S[PUSH(X);POP] = S
--|       );

end STACK ;
```

Appendix II
SYNTAX SUMMARY

The following is the syntax summary of the subset (refer to the Anna reference manual [2] for more details):

```
basic-declaration ::=
    ada_basic_declaration
  | basic-annotation-list

basic-annotation-list ::=
    basic-annotation {, basic-annotation};

basic-annotation ::=
    object-annotation
  | result-annotation

object-annotation ::=
    boolean-compound-expression
  | out boolean-primary

full-type-declaration ::=
    ada_full_type_declaration
    [ subtype-annotation ]

subtype-declaration ::=
    ada_subtype_declaration
    [ subtype-annotation ]

subtype-annotation ::=
    where identifier : type-mark => boolean-compound-expression;

name ::=
    ada_name
  | initial-name

compound_expression ::=
    expression [ implication-operator expression ]

relation ::=
    simple-expression { relational-operator simple-expression }
  | simple-expression [ not ] in range
  | simple-expression [ not ] in type-mark
  | simple-expression [ is not ] in range
  | simple-expression [ is not ] in type-mark

primary ::=
    ada_primary
  | conditional-expression
  | initial-expression
  | ( compound-expression )
```

```
implication-operator ::=
   -> | <->

conditional-expression ::=
   if condition then
      compound-expression
   {elsif condition then
      compound-expression}
   else
      compound-expression
   end if

condition ::=
   boolean-compound-expression

initial-name ::=
   in simple-name

initial-expression ::=
   in ( compound-expression )

simple-statement ::=
   ada_simple_statement
 | basic-annotation-list

compound-statement ::=
   [ compound-statement-annotation ]
   ada_compound_statement

compound-statement-annotation ::=
   with
      basic-annotation-list;

subprogram-declaration ::=
   ada_subprogram_specification;
   [ subprogram-annotation ]

subprogram-specification ::=
   ada_subprogram_specification
   [ subprogram-annotation ]

subprogram-annotation ::=
   where
      basic-annotation-list

result-annotation ::=
   return [ identifier : type-mark ❑ > ] compound-expression
```

Appendix III
AN ANNOTATED PROCEDURE TO EVALUATE P_n^r AND THE CORRESPONDING TRANSFORMED ADA PROCEDURE

```
with TEXT-IO; use TEXT-IO:

procedure MAIN is

   subtype INPUT-VALUE is INTEGER;
   --| where X:INPUT_VALUE =>
   --|     x <= 20;

   N,R:INPUT_VALUE := 0;

   package I_O is new INTEGER-IO(INTEGER);
   use I_O;

   function P( N, R: INTEGER) return INTEGER
      --I where
      --|     in (N >= R), in (N > 0), in (R >= 0);
   is

      RESULT:INTEGER := 1:

      --: function FACTORIAL( X: INTEGER) return INTEGER;

      -- | return FACTORIAL( N)/FACTORIAL( N-R) ;
      -- note: this is an annotation on P.

      --: function FACTORIAL( X : INTEGER) return INTEGER
      --| where return
      --|     if X = 0 then
      --|         1
      --|     else
      --|         X * FACTORIAL(X-1)
      --|     end if:
      --: is
      --:     FACT:INTEGER := 1;
      --: begin
      --:     for I in 2 . . X loop
      --:         FACT := FACT • I;
      --:     end loop:
      --:     return FACT ;
      --: end FACTORIAL;

   begin
      for I in 0 . . R-I loop
         RESULT := RESULT * (N-I);
      end loop:
      return RESULT;
   end P;
```

```
begin
   --| with
   a-|     N >= 0;
   loop
      PUT("Enter N ( 0 to quit ) : ");
      GET(N):
      exit when N = 0;
      PUT("Enter R :");
      GET(R);
      --I N > 0, R >= 0;
      PUT("The answer is : ");
      PUT(RESULT(N,R));
      NEW-LINE;
   end loop:
end MAIN;
```

After the transformations described in sections 5,6 and 7, the resulting program is shown below:

```
with ANNA ; use ANNA:

with TEXT-IO; use TEXT-IO;

procedure MAIN is

    subtype INPUT-VALUE is INTEGER;
    function "="(X,Y:STANDARD.MAIN.INPUT VALUE) return
        STANDARD.MAIN.INPUT-VALUE renames STANDARD."=";
    function LE(X,Y:STANDARD.MAIN.INPUT_VALUE) return
        STANDARD.MAIN.INPUT-VALUE renames "<=";
    --| where X:INPUT_VALUE =>
    --|     LE(X,20);
    -- The above annotation is illegal Anna since it does not occur
    -- directly after a type declaration. However, this is an intermediate
    -- program generated by the Transformer, and will be transformed
    -- furthur to a legal Ada program. The above annotation is still
    -- associated with the type INPUT-VALUE.

    N,R:INPUT_VALUE := 0;

    package I_O is new INTEGER-IO(INTEGER);
    use I_O;

    function P( N, R: INTEGER) return INTEGER is

        IN_EXP1 : constant STANDARD. BOOLEAN : = (N >= R) ;
        IN_EXP2 : constant STANDARD.BOOLEAN : = (N > 0) ;
        IN_EXP3 : constant STANDARD .BOOLEAN : = (R >= 0) ;
        -- | IN_EXP1,IN_EXP2,IN_EXP3;

        RESULT:INTEGER := 1;

        function FACTORIAL( X: INTEGER) return INTEGER;

        N1:STANDARD. INTEGER renames N;
        R1 : STANDARD. INTEGER renames R;
        function FACTORIAL1(X:STANDARD. INTEGER) return STANDARD. INTEGER
            renames FACTORIAL;
        function SLASH( X ,Y: STANDARD. INTEGER) return STANDARD. INTEGER
            renames "/" ;
        --| return RET1:STANDARD.INTEGER =>
        --|     STANDARD."/"(RET7,SLASH
        --|         (FACTORIAL1(N1)/FACTORIAL1(N1-R1)));

        function FACTORIAL( X: INTEGER) return INTEGER is

            X1:STANDARD.INTEGER renames X;
            function EQUAL(X,Y:STANDARD.INTEGER) return STANDARD. INTEGER
                renames "=";
            function MINUS(X, Y: STANDARD. INTEGER) return STANDARD. INTEGER
                renames "-";
```

```
        function FACTORIAL2(X:STANDARD. INTEGER) return STANDARD. INTEGER
            renames FACTORIAL:
        function STAR( X ,Y: STANDARD. INTEGER) return STANDARD. INTEGER
            renames "*";
        --| return RET2 : STANDARD. INTEGER =>
        --|     (
        --|         (EQUAL(X1,0) and then
        --|             STANDARD. "=" (RET2,1))
        --|      or else
        --|         (not EQUAL(X1,0) and then
        --|             STANDARD. "="(RET2,STAR(X1,FACTORIAL2
        --|                 (MINUS(X,1)))))
        --|     )

        FACT:INTEGER := 1;

    begin
    LOOP7 :
        for I in 2 . . X loop
            FACT := FACT * I;
        end loop LOOP7 ;
        return FACT ;
    end FACTORIAL;

begin
LOOP2 :
    for I in 0 . . R-l loop
        RESULT := RESULT * (N-I);
    end loop LOOP2;
    return RESULT ;
end P;

begin
BLOCK 1 :
    declare
        N2:STANDARD.INTEGER renames N;
        function GE1( X, Y: STANDARD. INTEGER) return STANDARD. INTEGER
            renames ">=";
        --| GE1(N2,0);
    begin
    LOOP3 :
        loop
            PUT( "Enter N ( 0 to quit ) : ");
            GET(N);
            if N = 0 then
                exit;
            end if;
            PUT("Enter R :");
        BLOCK2 :
            declare
                N3:STANDARD.INTEGER renames N;
                function GT( X ,Y: STANDARD. INTEGER) return STANDARD. INTEGER
                    renames ">";
                R2:STANDARD. INTEGER renames R;
                function GE2( X ,Y: STANDARD. INTEGER) return STANDARD. INTEGER
```

```
                renames ">=";
            --| out (GT(N3,0)),
            --I out (GE2(R3,0));
          begin
             GET(R);
          end BLOCK2 ;
          PUT("The answer is : ");
          PUT(RESULT(N,R));
          NEW-LINE;
       end loop LOOP3;
    end BLOCK 7 ;
end MAIN:
```

The fully transformed program is shown below:

```
with ANNA; use ANNA;

with TEXT-IO; use TEXT-IO;

procedure MAIN is

    subtype INPUT-VALUE is INTEGER:
    function "="(X,Y:STANDARD.MAIN.INPUT_VALUE) return
        STANDARD. MAIN. INPUT-VALUE renames STANDARD. "=";
    function LE( X ,Y :STANDARD .MAIN. INPUT-VALUE) return
        STANDARD.MAIN.INPUT_VALUE renames "<=";

    N,R:INPUT_VALUE := 0;
    CHECK-N: BOOLEAN := CHECK-EXP(LE(N.20));
    CHECK-R : BOOLEAN := CHECK-EXP(LE(R.20));

    package I_O is new INTEGER_IO(INTEGER);
    use I_O;

    function INPUT-VALUE-CHECKING-FUNCTION
        (X:STANDARD.MAIN.INPUT_VALUE) return STANDARD.MAIN.INPUT_VALUE is
    begin
        if not (LE(X,20)) then
            raise ANNA-EXCEPTION;
        end if ;
        return X;
    exception
        when ANNA-EXCEPTION =>
            REPORT-ERROR ;
        when ANNA-ERROR =>
            raise ;
        when others =>
            give-an-error-message ;
            raise ANNA-ERROR;
    e n d INPUT_VALUE_CHECKING_FUNCTION;

    function IS_IN_INPUT_VALUE_CHECKING_FUNCTION
        (X:STANDARD.MAIN.INPUT_VALUE) return STANDARD.BOOLEAN is
    begin
        return LE(X,20);
    exception
        when ANNA-ERROR =>
            raise ;
        when others =>
            give-an-error-message ;
            raise ANNA-ERROR;
    end IS_IN_INPUT_VALUE_CHECKING_FUNCTION;

    function P( N, R: INTEGER) return INTEGER is

        IN_EXP1: constant STANDARD.BOOLEAN := (N >= R);
        IN_EXP2: constant STANDARD .BOOLEAN : = (N > 0) ;
```

```
IN_EXP3 : constant STANDARD.BOOLEAN := (R >= 0);
CHECK_ANNO_1 :BOOLEAN : = CHECK-EXP( IN_EXP1 j;
CHECK-ANNO-2: BOOLEAN : = CHECK-EXP( IN_EXP2 ) ;
CHECK_ANNO_3 : BOOLEAN : = CHECK-EXP( IN_EXP3 );

RESULT:INTEGER := 1;

function FACTORIAL( X: INTEGER) return INTEGER;

N1 : STANDARD.INTEGER renames N:
R1 : STANDARD. INTEGER renames R;
function FACTORIAL1( X : STANDARD. INTEGER) return STANDARD. INTEGER
    renames FACTORIAL;
function SLASH(X ,Y : STANDARD. INTEGER) return STANDARD. INTEGER
    renames " / " ;

function CHECKING_FUNCTION_FOR_P( X: STANDARD. INTEGER)
        return STANDARD. INTEGER is
    RET7 : STANDARD, INTEGER renames X;
begin
    if not (STANDARD, " = " (RET1, SLASH
            (FACTORIAL1(N1)/FACTORIAL1(N1-R1)))) then
        raise ANNA-EXCEPTION:
    end if;
    return X;
exception
    when ANNA-EXCEPTION =>
        REPORT-ERROR ;
    when ANNA-ERROR =>
        raise ;
    when others =>
        give-an-error-message ;
        raise ANNA-ERROR;
end CHECKING_FUNCTION_FOR_P;

function FACTORIAL( X: INTEGER) return INTEGER is

    X1 : STANDARD. INTEGER renames X;
    function EQUAL ( X, Y : STANDARD. INTEGER) return STANDARD. INTEGER
        renames " = " ;
    function MINUS ( X ,Y: STANDARD. INTEGER) return STANDARD. INTEGER
        renames " - " ;
    function FACTORIAL2 ( X : STANDARD . INTEGER) return STANDARD. INTEGER
        renames FACTORIAL;
    function STAR ( X , Y : STANDARD. INTEGER) return STANDARD. INTEGER
        renames " * " ;

    FACT:INTEGER := 1;

    function CHECKING_FUNCTION_FOR_FACTORIAL
            (X : STANDARD. INTEGER) return STANDARD. INTEGER is
        RET2 : STANDARD. INTEGER renames X;
    begin
        if not (
                (
```

```
                    (EQUAL(X1,0) and then
                        STANDARD."="(RET2,1))
                 or else
                    (not EQUAL(X1,0) and then
                        STANDARD."="(RET2,STAR(X1,FACTORIAL2
                            (MINUS(X,1)))))
                )
            ) then
                raise ANNA-EXCEPT/ON;
            end if;
            return X;
        exception
            when ANNA-EXCEPTION =>
                REPORT-ERROR;
            when ANNA-ERROR =>
                raise ;
            when others =>
                give-an-error-message ;
                raise ANNA-ERROR;
        end CHECKING_FUNCTION_FOR_FACTORIAL ;

    begin
    LOOP1:
        for I in 2 . . X loop
            FACT : = FACT * I;
        end loop LOOP1;
        return CHECKING_FUNCTION_FOR_FACTORIAL( FACT) ;
    end FACTORIAL:

begin
LOOP2 :
    for I in 0 . . R-l loop
        RESULT := RESULT * (N-I);
    end loop LOOP2;
    return CHECKING_FUNCTION_FOR_P( RESULT) ;
end P;

begin
BLOCK 1:
    declare
        N2:STANDARD. INTEGER renames N;
        function GE7 ( X , Y : STANDARD. INTEGER) return STANDARD. INTEGER
            renames ">=" ;
        CHECK_ANNO_4: BOOLEAN : = CHECK-EXP( GE1( N2 , 0 ));

        function CHECKING_FUNCTION_FOR_N( X: STANDARD. INTEGER)
                return STANDARD. INTEGER is
            N2:STANDARD.INTEGER renames X;
        begin
            if not (GE1(N2,0)) then
                raise ANNA-EXCEPTION;
            end if;
            return X;
        exception
            when ANNA-EXCEPT/ON =>
```

```
            REPORT-ERROR ;
        when ANNA-ERROR =>
            raise ;
        when others =>
            give-an-error-message ;
            raise ANNA-ERROR;
    end CHECKING-FUNCTION-FOR-N;

begin
LOOP3 :
    loop
        PUT("Enter N ( 0 to quit ) : ");
        GET(N);
        N := CHECKING-FUNCTION-FOR-N
                (INPUT-VALUE-CHECKING-FUNCT/ON( N) ) ;
        if N = 0 then
            exit;
        end if;
        PUT( "Enter R : ");
    BLOCK2 :
        declare
            N3:STANDARD.INTEGER renames N;
            function GT( X, Y: STANDARD. INTEGER) return STANDARD. INTEGER
                renames ">";
            R2:STANDARD.INTEGER renames R;
            function GE2( X ,Y: STANDARD. INTEGER) return STANDARD. INTEGER
                renames ">=" ;
        begin
            GET(R);
            R := INPUT_VALUE_CHECKING_FUNCTION( R) ;
            CHECK_EXP(GT(N3,0)) ;
            CHECK_EXP(GE2(R3, 0)) ;
        end BLOCK2 ;
        PUT( "The answer is : ");
        PUT(RESULT(N,R));
        NEW-LINE;
    end loop LOOP3;
end BLOCK 7 ;
end MAIN;
```

References

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

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

[3] Evans, A., Butler, K.J., Goos, G., and Wulf, W.A.
DIANA Reference Manual, Revision 3
Tartan Laboratories Incorporated, 1983.

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

[5] Luckham, D.C., and von Henke, F.W.
An Overview of Anna - A Specification Language for Ada.
IEEE Software :9-22, March, 1985.

[6] Rosenblum, D.S.
A Methodology for the Design of Ada Transformation Tools in a DIANA Environment.
IEEE Software :24-33, March, 1985.

[7] Rosenblum, D.S., Sankar, S., and Luckham, D.C.
Concurrent Runtime Checking of Annotated Ada Programs.
Submitted to the Sixth Conference on Foundations of Software Technology & Theoretical Computer Science, New Delhi, India, December, 1986.

[8] Sankar, S., Rosenblum, D.S., and Neff, R.
An Implementation of ANNA.
Ada in use - Proceedings of the Ada International Conference :285-296, May, 1985.

[9] Sankar, S., and Rosenblum, D.S.
Transformation Methodology for Runtime Checking of Anna.
Submitted to IEEE Transactions on Software Engineering, March, 1986.

Index