Conversion of Fortran to Ada* using an Intermediate Tree Representation

John K. Slape and Peter J. L. Wallis
School of Mathematics, University of Bath, Claverton Down, Bath BA2 7AY, UK

We describe a Fortran to Ada converter that uses a standard intermediate tree representation of the program being converted. Points of some general interest include overall comments on high level language conversion and the selection of an intermediate representation as well as the way the design of this system was influenced by that of the interactive improvement system for which it is ultimately to provide components. Finally, the existing system and the conversions applied to some selected Fortran constructs are described in detail.

1. INTRODUCTION

The present system for the conversion of Fortran programs to Ada was initially developed on a mainframe computer, but is intended to provide components for the interactive Bath improvement system (IBIS), planned for use on Perq workstations. In describing the work, the selection of an intermediate representation for programs is considered before introducing the overall design of IBIS. Subsequent detailed discussion of the Fortran/Ada Converter addresses strategic issues arising from its eventual use within IBIS as well as details of the processing of selected Fortran constructs.

2. GENERAL BACKGROUND

In general, automatic high-level language conversion is not necessarily a viable way of introducing a change in working practices; in specific cases (for example, Ref. 1) it may be technically infeasible if the languages being converted are too radically different, or commercially unattractive if the majority of the users are likely, in any case, to rewrite most of their programs to enhanced specifications. Although there appears to be some potential for Fortran/Ada conversion of such specialized software as portable numerical libraries, the problem with such conversions is frequently that the generated programs are not sufficiently idiomatic in their use of the target language to make subsequent maintenance of the target language version an attractive proposition.

The present system initiates an attempt to improve this situation by use of an interactive program improvement system designed for a personal workstation. Existing language conversion systems (e.g. Refs 2–4) all use some combination of line-by-line conversion, program transformation and prettyprinting, but these different functions are frequently thoroughly intertwined. To allow flexibility in the choice of transformations applied to any particular program, these functions are quite deliberately separated in the present system by using a standard internal tree representation of the software being converted. This separation has the great advantage that the different transformations applied to programs being converted can be implemented as a collection of free-standing tools. The transformations to be applied to a particular piece of software can then be chosen not in a standard way, but under the control of a programmer using the system interactively. Thus we are concerned with the production not of a monolithic converter but rather with a collection of many small tools communicating through a common internal representation of programs, in accordance with the style of modern toolsets.[5,6]

* Ada is a registered trademark of the US Government, Ada Joint Program Office.

3. SELECTION OF AN INTERFACE

As in other projects involving many free-standing tools, and in program transformation systems in general, the viability of the whole project depends crucially on the program representation used for the internal interface. It was decided at an early stage that a tree representation was appropriate. Further, the interface should be Ada-related rather than Fortran-related; this allows later extension of the system for other purposes such as the improvement of Ada programs, as discussed below for IBIS. Thus it seems reasonable to use an attributed parse-tree representation of Ada programs as a basis for the internal representation, the use of attributes providing a mechanism for the sharing between tools of information about the program.

A standard for attributed Ada parse trees already exists in the Diana specification. Diana (descriptive intermediate attributed notation for Ada) is the specification of a standard abstract data type that implements attributed Ada parse trees. Since it is an abstract data type it is quite independent of the way the trees are actually represented; the Diana standard also includes a standard external representation of Diana structures as character strings for communication between different computing systems. Diana is maintained under contract to the US Department of Defense; the latest version of the *Diana Reference Manual* is Ref. 7 and the Diana design philosophy is the subject of Ref. 8.

3.1 Choice of Diana

Diana was chosen as the interface for the present project because of the availability of a complete, maintained standard as an abstract data type with an associated external representation. Further, the Diana standard

CCC-0010-4620/83/0026-0344$05.00

allows users to define additional attributes which could be used for communication between the program transformers we envisage, as mentioned earlier.

It was hoped initially that our choice of Diana might lead to free exchanges of tools with other parties and that we might find an implementation of Diana we could use. Diana is not currently sufficiently rigorously standardized to make these things possible. The problem is that the use of a general abstract data type interface within a compiler for a structure as pervasive as a parse tree introduces inefficiencies that can be overcome if assumptions are made about the tree representation (for example, by using Ada record selectors for accessing parts of the tree as in the York Ada compiler[5]). It follows that compiler-writers, by far the dominant current users of Ada parse trees, are naturally opposed to rigorous Diana standardization since they see this as compromising compiler efficiency. For tools, particularly on workstations, it seems better to keep the full generality of the abstract data type interface; thus we may presumably hope to see stronger standardization of Diana for tools than is the case at present. Given this situation, our philosophy has been to keep to the standard Diana abstract data interface as closely as possible, and to produce our own implementation of it. This implementation is modelled on the one developed for the Karlsruhe front-end.[9] There are several advantages in keeping as closely as possible to the Diana abstract data type specifications—most of the program is totally independent of the way Diana is represented and the rigorous use of the specification could only help us if it were necessary to produce a version of the system that uses a different standard attributed tree representation, such as the Ada Intermediate Representation (AIR) developed for the York Ada compiler.[5]

4. IBIS SYSTEM DESIGN

The present Fortran/Ada Converter is ultimately to provide components for the interactive Bath program improvement system (IBIS). IBIS is intended eventually to handle several different source and target languages via a common Diana interface. An overall view of the whole system is given by the data-flow diagram in Fig. 1. Here arrows represent tools transforming programs from one form to another; solid arrows denote existing components (although most transformers remain to be written) and broken arrows denote projected components, some of which might ultimately originate in other systems.

As shown in Fig. 1, IBIS could include an Ada front-end, allowing its use for transforming Ada programs under development. Source languages other than Fortran and Ada, and target languages other than Ada, could be handled also. However, the use of Diana does mean that any back-ends for languages other than Ada will only generate language constructs derivable from Diana in a similar manner to the PascAda back-end for Pascal.[10] Any combination of front- and back-end for which this turns out to be a serious restriction should presumably be using an intermediate form other than Diana and hence cannot be accommodated within the present system.

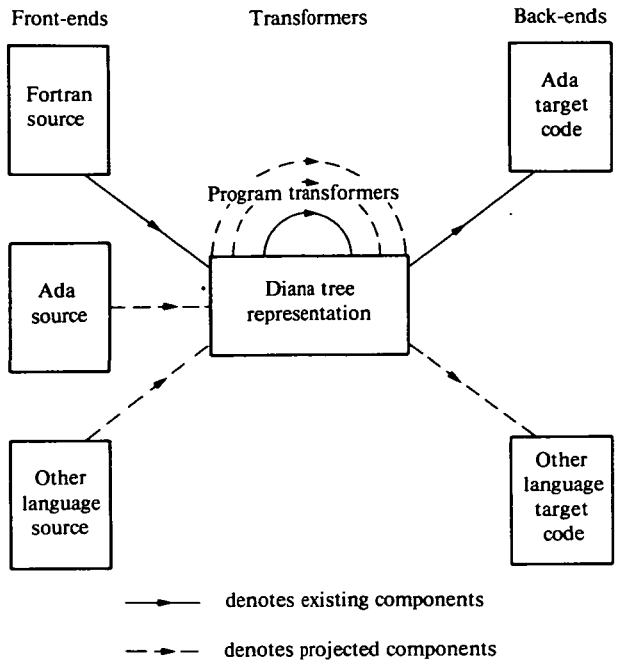

Figure 1. Overall organization of the IBIS system.

5. EFFECT OF IBIS ON THE CONVERTER DESIGN

As an introduction to the design of our Fortran/Ada Converter, we show how the requirements for IBIS have influenced the design of the current Fortran/Ada Conversion system. These considerations apply equally to any future front-ends and improvers written for IBIS. Back-ends are not considered here: the Diana-to-Ada prettyprinter (again based on one developed in connection with the Karlsruhe front-end[9]) is quite straightforward and need not detain us.

A given IBIS front-end processes input only from one source language, whereas the transformers are in general applicable to Diana originating from several languages. So it seems sensible *a priori* to keep front-ends as simple as possible, leaving any sophistication for the improvers. However, this requirement must be relaxed as necessary to conform with two others imposed by the need to keep the Diana general, namely:

(R.1) The Diana generated by a front-end should represent legal Ada as far as possible.
(R.2) Attributes added to the Diana used should not include any specific to particular source languages.

In the context of the Fortran/Ada conversion system (R.1) means for instance that the special processing for parameters that are subroutines or functions (discussed later) must be handled in the front-end rather than by using an improver that accepts 'Diana' corresponding to an imaginary Ada extension in which such parameters are allowed. However, it occasionally seems reasonable to allow a slight relaxation of (R.1) in cases where a straightforward Ada compilation error would clearly result if the condition went uncorrected. One such exception in the current front-end is made for extended-range DO-loops in standard Fortran. If we take no special actions over these (which is the appropriate action for a

simple, statement-by-statement front-end), we initially generate Diana in which labels are used from outside their scopes—this would raise an Ada compilation error.

In the current system (R.2) means that we disallow attributes corresponding to such conditions as 'statement arises from conversion of a Fortran DO-loop', or to the original Fortran types of variables. Thus a transformer for Ada generated from Fortran DO-loops, as discussed later, will process all instances of the resulting Ada structure; if a transformer were needed that explicitly processed *only* the Ada resulting from Fortran DO-loops, it should be included in a front-end. Similar remarks apply to transformers requiring knowledge of the original *source-code* types of variables; this may be necessary in some projected front-ends for languages other than Fortran or Ada.

It should be noted that the requirement (R.2) does not preclude passing parts of the original source text into the Diana as comments. To explain the handling of these by a front-end, we digress to consider how comments in general are handled by front-ends.

5.1 Handling of comments

Clearly, comments originating within the source text should be carried through to the generated Ada code. This is easily handled by storing comments in the node of the Diana tree that they reference. Thus a Fortran comment preceding an executable statement is stored in the Diana node representing the Ada corresponding to the statement; comments preceding Fortran specification statements such as COMMON, DIMENSION or EQUIVALENCE statements are all grouped together and stored in the Diana representation of the corresponding Ada declaration list. The nature of Fortran specification statements, allowing several items to be specified in one statement and particular items to be specified in more than one statement, makes it difficult to be more accurate with the placing of comments occurring within such statements. Proposed transformers include a special tool for repositioning comments.

A case can be made for generating further comments such as the original source-lines or comments on difficult conversions. These could be useful, for example, in the early stages of an interactive Fortran/Ada conversion when the user is still thinking in terms of the original Fortran. We include them, but they are distinguished from comments originating in the source text; this enables us to provide a transformer to remove them *en masse*.

5.2 Design criteria for front-ends

The use of front-ends within IBIS, as already explained, motivates the idea that in general these should be as simple as possible consistent with their performing all processing specific to the particular source-language. The main advantage of this way of doing things is that any program modification implemented by a transformer may be applied or not in a particular case at the discretion of the programmer using the interactive system. For example, it seems most appropriate to translate an arithmetic IF statement into a **case** statement with a **goto** statement in each alternative; a transformer might then dispose of one or more of the **goto**s by moving sections of code back into the arms of the **case**. Another example arises in the translation of a DO-loop into an Ada **for**-loop; since the Ada loop includes an implicit declaration of the control variable, the **for**-loop control variable cannot, in general, be used as the translation of the DO-loop variable but in certain special cases it can be. Such special cases could be detected by suitable transformers.

To summarize, the requirements for IBIS mean that the use of Diana should largely correspond to legal Ada, and that in general front-ends should be fairly naïve, statement-by-statement processors. This general philosophy for IBIS components applies particularly to the present Fortran/Ada Converter, described in the remaining Sections.

6. THE EXISTING SYSTEM

The Fortran/Ada conversion system at present comprises a front-end, a single modest improver and an Ada back-end as indicated by the solid arrows in Fig. 1. It runs on a Honeywell Multics system and is written in Pascal for subsequent transfer to a Perq computer. The system described here consists of about 9000 lines of Pascal source code.

The components of the system, together with their sizes in Pascal source lines and their interdependencies, are shown in Fig. 2. Tests of the system have included a conversion of a complete chapter of the NAG Library.

Running times for the current system as applied to a Fortran program of about 180 lines are shown in Table 1, which gives separate timings for runs with the Diana

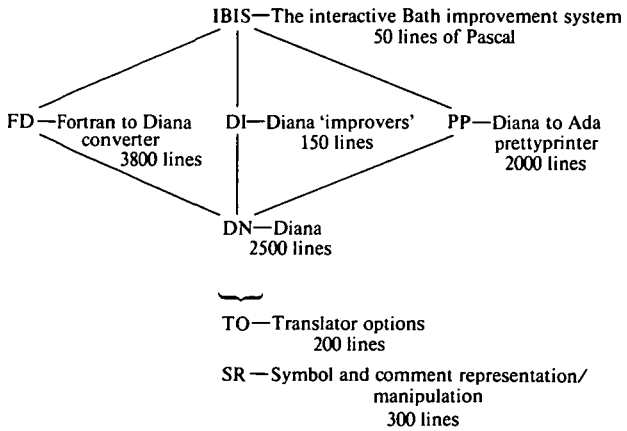

Figure 2. Fortran/Ada conversion system: module sizes and interdependencies.

Table 1. Multics processing times for Fortran program of 180 lines

Operation	CPU time, s
Fortran to Diana	8.1
Diana to filestore	3.1
Filestore to Diana	4.5
Diana loop improvements	0.9
Diana to filestore	2.7
Filestore to Diana	3.9
Prettyprinting	4.8
Compile original Fortran	2

in main store throughout and with writing of the Diana to filestore between the components of the system. These timings appear unfavourable compared with those of the Fortran compiler, which is highly optimized in terms of the use it makes of the Multics system facilities. It is believed that much of the running time of the current Converter is attributable to the completely procedural nature of the Diana interface, and to the amount of checking performed within the Diana procedures to ensure security of the Diana representation. The effects of these issues on the Converter running time are still being investigated.

An overall impression of the actions of the system in a simple case is provided by Figs 3–5. Figure 3 shows a listing of a small Fortran subroutine and Fig. 4 shows the generated Ada with the Fortran source statements included as comments. Finally, Fig. 5 shows the Ada resulting from application of the modest loop improver described in our later discussion of Fortran DO-loops; clearly further improvers to remove **null** statements and unused variables and labels could profitably be used.

```
      SUBROUTINE VECSCL ( VECTOR, VECSIZ, SCLFAC )
      INTEGER VECSIZ
      REAL VECTOR ( VECSIZ )
      DO 10 I = 1, VECSIZ
        VECTOR(I)= VECTOR(I)*SCLFAC
10    CONTINUE
      RETURN
      END
```

Figure 3. Small Fortran subroutine.

7. DETAILS OF THE CONVERTER

Remaining sections of this paper are devoted to detailed discussion of the design of the current Fortran to Ada Converter. Except where explicitly mentioned otherwise, we loosely refer to 'generated Ada' meaning the Ada listing resulting from application of the prettyprinter (back-end) to the Diana generated by the front-end with no program transformers used.

Progressing from global to specific issues, we first give details of the source and object languages supported. The structure of generated Ada programs is then discussed leading us on to consider communication between packages in the generated Ada. Finally selected Fortran statements, and the actions of a transformer which acts on Ada code generated from Fortran DO-loops, are discussed in detail.

7.1 Source and object languages

The source language for the current Converter is Fortran 77[11] or Fortran 66[13] according to the setting of a switch; this switch does not limit the language being converted

```
  with    FORTRAN_DATA_TYPES;
  use     FORTRAN_DATA_TYPES;
package   PACKAGE_VECSCL is
--      SUBROUTINE VECSCL ( VECTOR, VECSIZ, SCLFAC )
  procedure VECSCL (VECTOR : in out FREAL_ARR_DIM_1;
                    VECSIZ : in out FINTEGER;
                    SCLFAC : in out FREAL);
end PACKAGE_VECSCL;
  with    FORTRAN_DATA_TYPES, FORTRAN_ENVIRONMENT, FORTRAN_RUNTIME;
  use     FORTRAN_DATA_TYPES, FORTRAN_ENVIRONMENT, FORTRAN_RUNTIME;;
package   body PACKAGE_VECSCL is
--      SUBROUTINE VECSCL ( VECTOR, VECSIZ, SCLFAC )
  procedure VECSCL (VECTOR : in out FREAL_ARR_DIM_1;
                    VECSIZ : in out FINTEGER;
                    SCLFAC : in out FREAL) is
--      INTEGER VECSIZ
--      REAL VECTOR ( VECSIZ )
    I : F INTEGER;
    I_INCREMENT : FINTEGER;
  begin
--      DO 10 I = 1, VECSIZ
    I_INCREMENT := 1;
    I := 1;
    for I_CONTROL in 1..LOOP_COUNT (I, VECSIZ, I_INCREMENT) loop
--        VECTOR(I)= VECTOR(I)*SCLFAC
      VECTOR (I) := VECTOR (I) * SCLFAC;
--10    CONTINUE
    <<LABEL_10>>
      null;
      I := I + I_INCREMENT;
    end loop;
--      RETURN
    return;
--      END
    null;
  end VECSCL;
end PACKAGE_VECSCL;
```

Figure 4. The subroutine of Fig. 3 with Ada resulting from use of front-end.

```
with    FORTRAN_DATA_TYPES;
use     FORTRAN_DATA_TYPES;
package PACKAGE_VECSCL is
  procedure VECSCL (VECTOR: in out FREAL_ARR_DIM_1;
                    VECSIZ: in out FINTEGER;
                    SCLFAC: in out FREAL);
end PACKAGE_VECSCL;
with    FORTRAN_DATA_TYPES, FORTRAN_ENVIRONMENT, FORTRAN_RUNTIME;
use     FORTRAN_DATA_TYPES, FORTRAN_ENVIRONMENT, FORTRAN_RUNTIME;
package body PACKAGE_VECSCL is
  procedure VECSCL (VECTOR: in out FREAL_ARR_DIM_1;
                    VECSIZ: in out FINTEGER;
                    SCLFAC: in out FREAL) is
    I: FINTEGER;
    I_INCREMENT: FINTEGER;
  begin
    for I in 1 .. VECSIZ loop
      VECTOR (I) := VECTOR (I) * SCLFAC;
    << LABEL_10 >>
      null;
    end loop;
    return;
    null;
  end VECSCL;
end PACKAGE_VECSCL;
```

Figure 5. Ada from Fig. 4 after application of a modest loop-improver and removal of source statements.

but is used in the cases where the semantics of the languages differ such as zero-traversal DO-loops which are discussed later. There are several minor restrictions many of which are also imposed by PFORT,[12] a portable subset of Fortran. Restrictions worth noting are that unformatted input/output is not supported, formatted input/output is not currently supported and an assumption is made that Fortran functions do not change the values of their parameters. This last assumption, necessary because Ada functions can only have *in* parameters, cannot easily be checked by a Converter working on a line-by-line basis; violations result in Ada compilation errors. Analysis of the source code by the Converter starts with a classification of Fortran statements using an extension of Sale's Algorithm[14] to be documented elsewhere. Endeavours are being made to keep up with Ada changes as they occur; the latest version of the *Ada Language Reference Manual* is Ref. 15.

7.2 Structure of generated Ada

In considering the structure of generated Ada programs, a description of their overall structure in terms of Ada packages is followed by consideration of the library support they require.

7.2.1 Package structure of generated Ada. Before discussing the Ada package structure, the reader is reminded of the Fortran term 'program unit' which means a Fortran main program, subroutine, function or BLOCK DATA program unit. The program units constituting a complete program may be submitted to a Fortran compiler independently, if required, with inter-unit communication being resolved by a 'linkage editor' or 'linkage loader'.

Analogously, it was decided that the Converter should work on the basis of Fortran program units; a Fortran main program is converted to a corresponding Ada procedure and a Fortran subprogram (function or subroutine) 'NAME' is converted to an Ada package specification 'PACKAGE_NAME' containing a stub corresponding to the subprogram specification, and a package body containing the body of the subprogram. If a program unit 'NAME' then references another program unit 'NAME1' then 'PACKAGE_NAME1' will appear in the **with** and **use** statements of the converted form of 'NAME'. The conversion of BLOCK DATA program units is dealt with later when we discuss COMMON storage.

The alternative of requiring a complete Fortran program to be presented to the Converter and then producing a single Ada program was rejected for four reasons. First, our strategy is to keep the front-end as simple as possible: separate processing of program units means that the front-end is not concerned with inter-unit communication. Secondly, our approach is more in line with Fortran's independent compilation, the role of the 'link editor' being provided by the Ada compilation system. Thirdly, separate processing of program units enhances the attractiveness of the Converter for conversion of existing Fortran subroutine libraries. Fourthly, we envisage special IBIS tools for making changes to the package structure of Ada programs and it seems easier to design tools for amalgamating packages than for splitting them. The strategy adopted also has a further incidental advantage in that it allows us to handle Fortran COMMON storage in the way discussed below.

7.2.2 Run-time support for generated Ada. Three Ada packages will be provided for the run-time support of generated programs. Their names—'FORTRAN_DATA_TYPES', 'FORTRAN_ENVIRONMENT' and 'FORTRAN_RUNTIME'—appear, when required, in the **with** and **use** statements of each converted program unit. An outline of the package specification for 'FORTRAN_DATA_TYPES' is given in Fig. 6; this specifies types corresponding to the Fortran primitive types INTEGER, REAL, etc. and predefines arrays (up to seven dimensions) of these types so that arrays may be

```
package FORTRAN_DATA_TYPES is
  |
  type FINTEGER is
    range FINT_LOWER_BOUND..FINT_UPPER_BOUND;
  type FREAL is
    digits FREAL_DIGITS
    range FREAL_LOWER_BOUND..FREAL_UPPER_BOUND;
  type FDOUBLE_PRECISION is
    digits FDP_DIGITS
    range FDP_LOWER_BOUND..FDP_UPPER_BOUND;
  type FCOMPLEX is
    record
      REAL, IMAG: FREAL;
    end record;
  type FLOGICAL is new BOOLEAN;
  type FCHARACTER is new STRING;
  type FINT_ARR_DIM_1 is array ( INTEGER range <> ) of FINTEGER;
  type FINT_ARR_DIM_2 is array ( INTEGER range <>,
                                 INTEGER range <> ) of FINTEGER;
  |
  type FINT_ARR_DIM_7 is...
  -- same for arrays of FREAL, FDOUBLE_PRECISION, FCOMPLEX
  --   and FLOGICAL
  type FCHAR_ARR_DIM_1_LEN_1 is array ( INTEGER range ;<> )
    of FCHARACTER(1..1);
  type FCHAR_ARR_DIM_1_LEN_2 is array ( INTEGER range <> )
    of FCHARACTER(1..2);
  |
  type FCHAR_ARR_DIM_1_LEN_256 is array ( INTEGER range <> )
    of FCHARACTER(1..256);
  |
  type FCHAR_ARR_DIM_7_LEN_256 is...
end FORTRAN_DATA_TYPES;
```

Figure 6. Outline of the package FORTRAN_DATA_TYPES.

passed as parameters in the converted programs. Under a previous edition of the *Ada Language Reference Manual*[18] it was possible to specify types corresponding to the Fortran CHARACTER type and arrays of CHARACTER type as in Fig. 7; this involved a small fixed number of types rather than the cumbersome number of types now required, which is dependent on the maximum length of Fortran CHARACTER types we allow to be converted (Fig. 6 limits this to 256 characters). The package 'FORTRAN_ENVIRONMENT' specifies the intrinsic functions available in Fortran and various operators for supporting the types defined in 'FORTRAN_DATA_TYPES', and the package 'FORTRAN_RUNTIME' provides all other facilities assumed available by the converter including support for Fortran formatted input/output.

7.3 Communication between packages

Next we discuss some Converter actions that affect communication between packages in the generated Ada.

```
type FCHARACTER ( LENGTH: FCHAR_RANGE ) is
  record
    FCHAR: STRING(1..LENGTH);
  end record;
type FCHAR_ARR_DIM_1 ( LB1, UB1: INTEGER,
                       LENGTH: FCHAR_RANGE) is
  record
    FCHAR: array(LB1..UB1) of STRING(1..LENGTH);
  end record;
type FCHAR_ARR_DIM_2 ( LB1, UB1,
                       LB2, UB2: INTEGER,
                       LENGTH: FCHAR_RANGE ) is
  record
    FCHAR: array(LB1..UB1, LB2..UB2) of STRING(1..LENGTH);
  end record;
  |
type FCHAR_ARR_DIM_7...
```

Figure 7. Extract from a previous version of the package FORTRAN_DATA_TYPES.

These comprise the treatment of Fortran COMMON storage and BLOCK DATA program units, the EQUIVALENCE statement and various issues related to the conversion of parameters.

7.3.1 Fortran COMMON storage and BLOCK DATA program units. The specification of a shared area of storage in Fortran, COMMON, must be given by each program unit that uses the shared area, and it is possible for different program units to interpret the area in different ways. In the current Converter, it is assumed that positionally matching variables in the different specifications of a particular COMMON block agree in type and size of dimension, although they may have different names. Based on this assumption, the way in which the Converter treats COMMON statements is dependent on the type of the program unit in which they appear, as discussed next.

The Converter treats the specification of a COMMON block appearing in a BLOCK DATA program unit (which is used for initializing COMMON blocks) as the definitive specification for that COMMON block. Each COMMON block specified in a BLOCK DATA program unit is converted to an Ada package specification. For example, the BLOCK DATA program unit

```
BLOCK DATA
COMMON /FRED/ A, B
DATA A/0.0/, B/0.0/
END
```

is converted to just one Ada package

```
   with FORTRAN_DATA_TYPES;
   use FORTRAN_DATA_TYPES;
package COMMON_FRED is
   NAME_1: FREAL := 0.0;
   NAME_2: FREAL := 0.0;
end COMMON_FRED;
```

References made to a COMMON block in other types of program unit are then converted to a sequence of Ada **rename** statements. For example, the Fortran COMMON statement.

```
COMMON /FRED/ X, Y
```

appearing in a Fortran main program or subprogram, is converted to the Ada statements

```
X: FREAL renames COMMON_FRED. NAME_1;
Y: FREAL renames COMMON_FRED. NAME_2;
```

To complement the above approach in the cases where a particular COMMON block does not occur in a BLOCK DATA program unit, the Converter provides a method whereby the user may specify that the current program unit being converted (not a BLOCK DATA program unit) is to be taken as the definitive specification for a particular COMMON block. In such cases, as well as the **rename** statements being generated in the converted program, a separate package specification corresponding to the COMMON block is also generated. A particular case where this technique would have to be used is for Fortran's 'blank' COMMON block which cannot be initialized in a BLOCK DATA program unit. It is possible for blank COMMON blocks within an executable program to be of different sizes; in such cases the user should specify the program unit with the largest blank COMMON block to be the definitive specification.

Many Fortran programs use COMMON storage in a way that does not comply with the assumptions currently made by the Converter. For example, a common practice in Fortran programming is to put together a number of variables in a COMMON block, used in several sub-programs, but to replace the group by a suitably dimensioned dummy array in other sub-programs not using these variables. Such use will cause the Converter to generate a **rename** statement associating variables of different types, giving rise to an Ada compilation error. If the dummy array is not used within the offending program unit and is the last entry in the COMMON list (as is common practice to match COMMON block sizes), the offending **rename** statement may simply be removed. In more complicated cases, it should be possible to design IBIS tools to help the user rectify the resulting situation.

7.3.2 Fortran EQUIVALENCE statement. We briefly mention the EQUIVALENCE statement here, since it is possible to equivalence variables with other variables that are in COMMON blocks, and thus in a sense equivalencing can affect package communication. In general the problem of translating Fortran's EQUIVALENCE statement which equivalences storage sequences rather than variables of the same type and size, is not feasible. The present Converter treats EQUIVALENCE statements as untranslatable. In a later version of the Converter, variables with the same type and size will be converted using the Ada **rename** statements. Other equivalenced variables will be treated as totally separate entities; of course warning messages will be produced. A user of IBIS might be able to rectify the resulting situation if equipped with special tools for systematically renaming quantities throughout a package.

7.3.3 Ada parameter modes. In Fortran, parameters to both function and subroutine subprograms are passed using a combination of call-by-reference and call-by-value-result; whether or not a formal argument may be defined or redefined within a subprogram depends in general on the actual parameter. For example, a formal parameter associated with a constant expression cannot be redefined within the subprogram; however, if the same formal parameter were associated with a variable then it could be defined/redefined. For this reason it is impossible to calculate in all cases the modes of formal parameters when converting program units on an independent basis.

Appropriate Converter processing of formal parameters depends on whether these appear in a FUNCTION or a SUBROUTINE in the original Fortran. All formal parameters of Ada functions must be of the **in** mode, and since Fortran functions are converted to Ada functions the same is expected of the Fortran functions we translate. If a function formal parameter appears on the left hand side of an assignment statement within the function body being converted, a warning message is produced; the same is true if the formal parameter is used as an actual parameter to another subprogram (the intention is not determinable on an independent conversion basis). The alternative is to convert Fortran functions to Ada procedures and use auxiliary variables. This seems too awkward for general use, but it seems appropriate to provide an IBIS tool for doing it in specific cases.

The situation is different for formal parameters of a Fortran SUBROUTINE. The most general approach to converting these under the independent translation scheme is to convert the formal parameters of all subroutines to Ada *in out* mode parameters and to assign the actual parameters of subroutines which are constants or expressions (not simply variable names) to auxiliary variables, replacing the actual parameters with these variables.

Another possible approach might be to convert formal parameters of subroutines according to how they are used within the subroutine body—read, written or both. This involves putting some default interpretation on formal parameters used as actual parameters to other subprograms and hence is not a suitable approach for a front-end that treats different Fortran program units quite independently.

7.3.4 Procedures as parameters. Ada does not allow the passage of parameters to procedures that are themselves procedures, but enables a similar effect to be achieved through the use of generics. It turns out to be feasible to handle Fortran procedure parameters automatically in this way provided that it is possible to determine in a given program unit whether the procedure parameter is a function or a subroutine and to determine the types of its parameters; that is to say, a formal parameter that is a procedure must not simply be used as an actual parameter to another procedure; a warning is produced if this is done. Figure 8 outlines the conversion into Ada of a Fortran SUBROUTINE with a FUNCTION parameter.

7.3.5 Arrays as parameters. Arrays that appear as formal parameters are converted to the appropriate unconstrained array type defined in the package FORTRAN_DATA_TYPES, which we have previously discussed. This means that the bounds are obtained from the actual parameter and the formal parameter is constrained by these bounds. This is a particular case where it is necessary for the user to obey certain restrictions which cannot be checked using the independent conversion approach and could otherwise result in the generation of Ada programs which do not conform to the original Fortran program, for example, the case where Fortran 77 corresponding formal and actual array parameters are of the same size but have different origins. To help the user to rectify such cases and also cases where the number and size of dimensions of actual and formal parameters disagree,[16] which is also quite legal in Fortran, we anticipate using IBIS tools applicable to complete packages that translate multidimensional array references to single dimension references and for shifting the origins of dimension bounds.

Note in passing that a tool acting on a complete package to change multidimensional array references to single dimension references could have a further use in connection with the production of Ada programs to run on paging systems. The Ada standard does not specify whether multidimensional arrays are stored in row-major or column-major form, and this can make a considerable difference to the performance on paging systems of programs handling such arrays.[17] By using a tool that converts multidimensional array references to vector references, using row- or column-major form at the

```
      PROGRAM MYPROG
      EXTERNAL FUNC
       |
      CALL SIMP ( FUNC, A, B, N )
       |
      END
      SUBROUTINE SIMP ( FOFX, XO, XN, N )
       |
      Y = FOFX ( PT )
       |
      END
with    FORTRAN_DATA_TYPES, FORTRAN_ENVIRONMENT, FORTRAN_RUNTIME,
        PACKAGE_SIMP, PACKAGE_FUNC;
use     FORTRAN_DATA_TYPES, FORTRAN_ENVIRONMENT, FORTRAN_RUNTIME,
        PACKAGE_SIMP, PACKAGE_FUNC;
procedure MYPROG is
  |
  procedure SIMP_FUNC is new SIMP ( FUNC );
  |
begin
  SIMP_FUNC (A, B, N);
  |
end MYPROG;
  with    FORTRAN-DATA_TYPES;
  use     FORTRAN_DATA_TYPES;
package   PACKAGE_SIMP is
  generic
    with function FOFX ( PARAM_1 : in FREAL ) return FREAL is <>;
  procedure SIMP ( XO : in out FREAL;
                   XN : in out FREAL;
                   N : in out FINTEGER);
end PACKAGE_SIMP;
```

Figure 8. Outline of converter actions for Fortran program units with procedure parameters.

discretion of the user, a programmer could optimize performance in this way regardless of which method of array storage is used by his particular Ada compiler.

7.4 Fortran statements

Given the preceding work on program structure and communication between packages, the conversion of most Fortran statements is too straightforward to warrant detailed comment. We discuss briefly our treatment of Fortran declarations, assigned GOTO statements and DO-loops.

7.4.1 Declarations. The declaration of the attributes of a variable in Fortran may be spread over several statements; for example, the same variable could occur in DIMENSION, TYPE and COMMON statements. Variables may also be implicitly declared by their appearance in the executable part of the program text, in which case they acquire default attributes. For these reasons, the Converter does not translate Fortran specification statements directly into Diana, but builds a symbol table, which has been designed using Diana attributes, to which are also added auxiliary declarations required by the conversion process. This symbol table is subsequently used at the end of each program unit to generate the Diana form of the necessary declaration. Since the Diana for a complete program unit is held in main storage, this is compatible with the single-pass, line-by-line approach.

7.4.2 ASSIGN statement and assigned GOTO statement. The Fortran ASSIGN statement

ASSIGN *s* to *i*

where *s* is a statement label and *i* is an integer variable, is converted to the Ada statement

*i*_ASSIGN := LAB_*s*;

The Fortran assigned GOTO statement

GOTO *i* {,} (*s*{, *s*} . . .)

where *i* is an integer variable name and (*s*{, *s*} . . .) is a list of statement labels which must always be present (this restricts the Fortran 77 assigned GOTO statement where the list is optional), is converted to the Ada statement

```
case i_ASSIGN is
    when LAB_s1 ⇒ goto LABEL_s1;
    when LAB_s2 ⇒ goto LABEL_s2;

    when LAB_sN ⇒ goto LABEL_sN;
    when others ⇒ raise FT_ASSD_GOTO_EXCEP;
end case;
```

where *i*_ASSIGN is defined as type ASSIGNED_LABEL in the Ada program and the type ASSIGNED_LABEL is defined in the program as an enumeration of all the labels that appear in ASSIGN or assigned GOTO statements. The exception FT_ASSD_GOTO_EXCEP is defined in the package FORTRAN_RUNTIME.

The integer variable *i* (not *i*_ASSIGN) still exists in the program as an integer variable. This is necessary as Fortran allows variables used in ASSIGN statements and assigned GOTO statements to be used also as ordinary integer variables in other parts of the program. The Fortran rules for the variable being defined/undefined as integers or statement labels allow us to separate its use in this way.

7.4.3 Fortran DO-loops. The Fortran DO-loop construct

```
    DO s v=e1, e2, e3
    :
s   stm
```

where *s* is a statement label, *v* an integer, real or double precision variable and *e*1, *e*2 and *e*3 are integer, real or double precision expressions, is converted to the Ada statements:

```
v_INCREMENT := e3;
v := e1;
for v_CONTROL in 1 . . . LOOP_COUNT(v, e2,
v_INCREMENT) loop
    :
    ≪LABEL_s≫
      stm
      v := v + v_INCREMENT;
end loop;
```

where the variable *v*_INCREMENT is defined within the program and has the same type as *v*. The overloaded function LOOP_COUNT is defined in the package FORTRAN_RUNTIME and returns an integer value which is the maximum of the values INT((*e*2 − *v* + *v*_INCREMENT)/*v*_INCREMENT) and 0. This ensures a correct Ada interpretation of the Fortran 77 DO-loop.

To ensure that the loop is always executed at least once in the case of Fortran 66 the function call

LOOP_COUNT (*v*, *e*2, *v*_INCREMENT)

is replaced by the function call

LOOP_COUNT_OR_1(*v*, *e*2, *v*_INCREMENT)

which returns the maximum of the values INT ((*e*2 − *v* + *v*_INCREMENT)/*v*_INCREMENT) and 1.

The above approach ensures that the most general cases are catered for. In many cases, however, the generated Ada could be improved, for example:

(D. 1) If *e*3 is not present (i.e. assumed to be 1) or *e*3 is a constant expression, the variable *v*_INCREMENT need not be introduced—instead it may be replaced by the constant expression.

(D. 2) If the loop variable is of integer type, and the increment value is 1, then the loop may be 'collapsed' to

```
for v in e1 . . e2 loop
    :
    ≪LABEL_s≫
      stm
end loop;
```

provided that the value of the loop variable is not required after termination of the loop.

(D. 3) If there is no transfer of control from within the loop to the loop terminating label, then this label may be removed.

Comparison of Figs 4 and 5 show the effect of applying a simple program transformer that implements (D. 2). As mentioned previously, the resulting program could clearly be improved further, for instance by applying transformers that implement (D. 3) and that remove **null** statements.

8. CONCLUSION

The decision to use an intermediate tree representation in our Fortran/Ada Converter enables us to use it as the basis for an interactive program improvement system that may ultimately be used for many different purposes. Design considerations for IBIS resulted in our handling the specifically Fortran-dependent parts of our conversion in a fairly naïve statement-by-statement front-end that handles separate Fortran Program Units quite independently. Future plans include attempts to discover experimentally which improvers prove most useful in the interactive conversion of existing Fortran libraries to idiomatic Ada versions.

Acknowledgements

We are most grateful to the UK Science and Engineering Research Council for its support of John Slape's work on the Converter under Grant GR/B 45139, and for its subsequent support of IBIS under Grant GR/C 09319. Our thanks also to John Barnes, Bernd Krieg-Brückner, Brian Maher, Ian Pyle, Colin Runciman, George Symm, Brian Wichmann and Georg Winterstein for useful discussions, and to the Referee for suggested improvements.

REFERENCES

1. SPL International, A feasibility study of the conversion of RTL/2 to Ada. Technical Guide (September 1982) and Managerial Report (October 1982).
2. R. A. Freak, A Fortran to Pascal translator. *Software—Practice and Experience* **11**, 717–732 (1981).
3. K. Dickson, Translating Fortran 66 into Ada. *Final Year Project*, Department of Computer Science, University of Strathclyde (1981).
4. A. Prudom, The automatic translation of Fortran to Algol 68. *Ph.D Thesis*, Department of Computational and Statistical Science, University of Liverpool (n.d.).
5. I. C. Pyle, Private Communications.
6. L. J. Osterweil, *The Toolpack Mathematical Software Development Environment*, Department of Computer Science, University of Colorado at Boulder (July 1982).
7. A. Evans and K. J. Butler (Eds), *Diana Reference Manual, Revision 3*. Tartan Laboratories Incorporated (February 1983).
8. K. Butler and A. Evans, *Interim Diana Report*. Tartan Laboratories Incorporated (October 1982).
9. G. Goos and G. Winterstein, Towards a compiler front-end for Ada. In *Proceedings of the ACM SIGPLAN Symposium on the Ada Programming Language*, Boston (1980). *ACM SIGPLAN Notices* **15**(12), 36–46 (1980).
10. P. F. Albrecht, P. E. Garrison, S. L. Graham, R. H. Hyerle, P. Ip and B. Krieg-Brückner, Source-to-source translation: Ada to Pascal and Pascal to Ada. In *Proceedings of the ACM SIGPLAN Symposium on the Ada Programming Language*, Boston (1980). *ACM SIGPLAN Notices* **15**(12), 183–193 (1980).
11. American National Standards Institute Inc. *ANSI X3. 9–1978 Programming Language Fortran* (1978).
12. B. G. Ryder, The PFORT verifier. *Software—Practice and Experience* **4**, 359–377 (1974).
13. American National Standards Institute Inc. *ANSI X3. 9–1966 American National Standard Fortran* (1966).
14. A. H. J. Sale, The classification of Fortran statements. *The Computer Journal* **14** (1), 10–12 (1970).
15. United States Department of Defense. *Reference Manual for the Ada Programming Language ANSI/MIL-STD 1815 A* (January 1983).
16. A. H. Morris Jr, Can Ada replace Fortran for numerical computation? *ACM SIGPLAN Notices* **16**(12), 10–13 (1981).
17. G. M. Weinberg, Programming and compiling strategies for paging systems. *Software—Practice and Experience* **2**, 165–171 (1972).
18. United States Department of Defense. *Reference Manual for the Ada Programming Language* (July 1980).

Received January 1983