ADA AND ITS IMPACT ON THE SCIENTIFIC USER

R. Marty,
Institut für Informatik, Universität Zürich, Switzerland.

ABSTRACT

The Ada programming language is the result of a collective effort to design a common language for programming real-time systems. The design of Ada was initiated by the United States Department of Defense (DoD) in 1975. Ada combines facilities found in most classical languages like Fortran, PL/I, Pascal, and Basic together with many features formerly found only in experimental languages. It is argued that these features make Ada a very decent tool not only for writing real-time programs but also for the development of software in the scientific sector.

1 INTRODUCTION

The DoD is the largest software consumer on earth. As early as in 1970, it became apparent to DoD officials that a new standardized programming language should be defined and implemented to break the sharply rising trend of the exorbitant software costs. The requirements for such a standard language were worked out by DoD's High Order Language Working Group. 17 proposals for the new language were received and four of these were selected to compete for the final choice. The "Green" language, submitted by CII Honeywell Bull of France, made the race and was accepted as the new standard language for embedded computer systems programming. By this time, "Green" was renamed "Ada" in honour of the Countess of Lovelace, daughter of Lord Byron, who was Babbage's assistant and the world's first programmer.

The key features of Ada are:

- data abstraction (specification of abstract properties of data and not of physical representations, new types may be defined together with the operations on them)
- strong typing (every data object is bound to an abstract type, type incompatibilities are detected)
- block structure (hierarchical levels of scope for named objects)
- exception handling (mainly for user controlled error recovery)
- tasking (creation, deletion, and synchronization of parallel processes, interprocess communication)
- generic units (program units can be parametrized over appropriate data types, especially useful for library routines)
- programming in-the-large (encapsulation mechanisms, type-safe separate compilation, overloading of operators and subroutines, library mechanisms are part of the language definition)

The history of Ada's development clearly shows that Ada is a language which is to be used in programming embedded systems applications, i.e. for real-time programs. It is the intent of this paper to argue that Ada is also a very decent language for scientific programming, an area now largely dominated by the antiquated programming language FORTRAN, a language that fails to meet several paramount aspects of modern software engineering.

For the layout of this paper, measuring Ada against aspects of modern software engineering has given preference over an enumeration of Ada's features. Ada will not be discussed in its full scope here. We are only trying to give a first insight into the power of Ada. A

more detailed overview of Ada is given by Barnes in [1]. The book "Ada" by Springer-Verlag [2] contains an introduction to Ada and the Ada Reference Manual. Numerous papers on Ada may be found in recent issues of the ACM SIGPLAN Notices, especially in the November 1980 issue.

2 DATA ABSTRACTION

Programming a computer is always based on some mental model of computation residing in the programmer's brain. It is the programmer's task to map this abstract model of computation into the real model of computation given by the programming language used. The wider the gap between the abstract model and the programming language, the more cumbersome the programmer's work gets, and the more difficult will it be for a third person to remap the program into his or her own model of computation. The costs of maintaining software are significantly influenced by the difficulty of this process. A programming language should match an abstract model of computation as closely as possible.

A particularly well structured model of computation emerged from the Algol school and found its implementation in e.g. Algol 60, Algol 68, PL/I, Pascal, and Ada. Since the Algol model of computation found wide acceptance in education and practice and was the core of the bulk of algorithmic programming languages introduced in the last 15 years (including Ada), it seems reasonable to adapt it for scientific programming as well. Consequently, the programming language we choose should match this model as closely as possible, a requirement that is not met by FORTRAN, COBOL, or BASIC.

A program describes actions (or algorithms) to be performed on data. This leads to a discussion of Ada's abstraction mechanisms under the two headings "Data Abstraction" and "Algorithmic Abstraction". The former will follow right here, the latter in Chapter 3.

2.1 Numeric Types

The guiding principle of describing data in Ada is that a data description should define the abstract properties of the data object without any reference to a specific representation in hardware. As a consequence, the introduction of a new type for representing integer values includes an indication of the range of values to be represented by objects of this integer type:

```
type DAY   is range 1..31;
type TEMP  is range -50..+50;
type LIRE  is range -999999999..+999999999;
```

(It is just a notational convention to write keywords in small letters and identifiers in capitals). Objects of such a type may then be created by an object declaration:

```
TODAY     : DAY;                   -- uninitialized variable
AMOUNT    : LIRE := 0;             -- initialized variable
LIMIT     : constant TEMP := 27;   -- constant
```

An important advantage of explicitly defining the desired range is the independence from specific internal representations of integer values, i.e. from the hardware range of integers. This means that a language violation is detected immediately upon an integer value falling outside the declared range and not only upon producing a hardware overflow/underflow

condition. Giving explicit ranges of integer values also allows the Ada implementation to choose whatever internal representation for the integer object as long as the language semantics hold. The type LIRE above could for example be represented as a double word or in floating-point format. It is very important, however, that the programmer is and should be unanware of such implementation considerations.

The fixed-point types and the floating-point types are described in the same spirit of data abstraction as discussed for integers:

```
type HOURS        is delta 0.01  range -9999.99..+9999.99;
type WEIGHT       is delta 0.005 range 0..1000;
type MASS         is digits 7    range 0.0..1.0E10;
type COEFFICIENT  is digits 10   range -1.0..+1.0;
```

Besides the range, the accuracy of fixed-point types is specified by an absolute value, called the "delta" of the fixed-point type (the difference between two adjacent values of this type). The accuracy of floating-point types is given by indicating the minimum number of decimal digits for the decimal mantissa.

2.2 Enumeration Types

A very common use of integers is for designating elements of a set of nonnumeric values. Ada's so called "enumeration type" is a natural construct for this purpose:

```
type WEEKDAY         is (MON,TUE,WED,THU,FRI,SAT,SUN);
type MARITAL_STATUS  is (SINGLE,MARRIED,DIVORCED);
type DEPT            is (FOOD,HOUSEHOLD,STATIONERY,TOOLS);
```

The identifiers specified for an enumerated type are the ordered values of this type and are used as constants in expressions of this type:

```
TODAY : WEEKDAY;
  :
  :
if TODAY=SUN then ...
TODAY := MON;
```

The advantage of using enumerated types over encoding the corresponding information by integer values lies in the better documentation of the purpose of such a data object and in the enforced type safety.

2.3 Other Scalar Types

Scalar types are types whose values have no components. In addition to the scalar types introduced above, Ada defines a scalar type for logical data objects (data objects that can assume the values TRUE and FALSE) and one for character data objects. It is further possible to define subtypes of scalar types:

```
subtype WORKING_DAY  is WEEKDAY range MON..FRI;
subtype ALPHA        is CHARACTER range 'a'..'z';
```

2.4 Composite Types

A composite type is a type that is made up of components of other types. Ada defines the "array" and the "record" as composite types. The index type(s) of an array can be any discrete scalar type, e.g.:

```
type HOURS_WORKED   is array (WEEKDAY) of HOURS;
type MATRIX         is array (1..50,1..50) of range 0..1000;
type SOLD           is array (DEPT,WEEKDAY) of FRANCS;
```

Ada "records" are similar to COBOL and PL/I data structures:

```
type TIME is record
                 HH       : INTEGER range 00..23;
                 MM       : INTEGER range 00..59;
             end record;

type FLIGHT is record
                   AIRLINE   : STRING(2);
                   FLIGHT    : INTEGER range 0001..9999;
                   ARRIVAL   : TIME;
                   LANDED    : BOOLEAN;
               end record;
```

It is also possible to overlay the same storage space with different variants of records.

2.5 Access Types

Ada's access types correspond to pointers in PL/I and Pascal, a concept unimplemented in FORTRAN. In contrast to PL/I pointers but in accordance with Pascal, an Ada access type is bound to a specific type to prevent the pitfalls of unbound pointers:

```
type LINK is access FLIGHT;
```

Access types permit the building of dynamic data structures of almost arbitrary shape and complexity (linked lists, trees, graphs, etc.).

3 ALGORITHMIC ABSTRACTION

The chapter on data abstraction dealt with the abstract model of computation given by Ada as viewed from the description of data objects. This chapter introduces Ada's way of describing actions, i.e. the algorithmic abstraction as implemented in Ada.

3.1 Description of Expressions

An expression is a construct to create a new data object through application of operations on existing data objects. Examples for Ada expressions are:

```
AMOUNT / 100 * INTEREST_RATE / 360 * NO_OF_DAYS
(DEBIT-CREDIT)*FACTOR >= LOAN
MONTH not in JUN..SEP and TODAY in WORKING_DAY
FIRST_NAME & " " & LAST_NAME
```

Ada defines the following operators:

arithmetic:	+, -, *, /, mod (modulo division), rem (remainder), ** (exponentiation)
relational:	=, /= (not equal), <, >, <=, >=

membership test: in, not in
logical: and, or, xor (exclusive or), not

3.2 Flow of Control

The usual flow of control in an Ada program unit is sequentially from top to bottom. An *if* statement selects for execution one or none of a number of sequences of statements, depending on the truth value of one or more corresponding conditions:

```
if A<0 then      if A<0 then      if A<0 then          if A<0 then
    :                :                :                    :
end if;          else             elsif A>100 then     elsif A>100 then
                     :                :                    :
                 end if;          elsif A>10 then      elsif A>10 then
                                      :                    :
                                  end if;              else
                                                           :
                                                       end if;
```

A second statement for specifying conditional execution of sequences of statements is the *case* statement, roughly corresponding to PL/I's SELECT:

```
case TODAY is
    when MON        =>  :
    when TUE..FRI   =>  :
    when others     =>  :
end case;
```

Repetition of a sequence of statements is described by a *loop* statement in one of the following forms:

```
loop            while A<0 loop       for I in 1..100 loop
  :                 :                    :
end loop;       end loop;            end loop;
```

The leftmost form specifies an endless loop. Naming of loops is also possible. Any loop can be terminated by the execution of an *exit* statement.

Ada's statements used to describe the flow of control within a program unit are actually equivalent to many graphic notions (Jackson Diagrams, Nassi-Shneiderman Charts, etc.) used to design and document programs written in languages with poor control statements. Using such diagrammatical aids for designing and describing Ada programs is unnecessary since the programming language Ada itself can very well be used for this purpose during all stages of program development, from global layout right down to the final form of the program.

3.3 Subprograms

A subprogram is an executable program unit that is invoked by a subprogram call. There are two forms of subprograms: procedures and functions. A procedure call is a statement, a func-

tion call returns a value.

In Ada every formal parameter of a procedure or function has a mode:

- in parameters act as inputs to the subprogram
- out parameters act as outputs from the subprogram
- inout parameters act as variables whose values may be updated during execution of the subprogram

Formal parameter specifications may also include default values which are used when the corresponding formal parameter is not present in the call:

```
procedure PRINT_HEADER ( PAGES   : in INTEGER := 1;
                         HEADER  : in LINE;
                         CENTER  : in BOOLEAN := FALSE);
```

All Ada subprograms are reentrant and can be called recursively.

A suitable subprogram concept is a very important corner stone in modern software engineering. It allows programming on different levels of abstraction by hiding internal mechanisms and only interacting with the environment via a well defined interface. This leads to simpler and better understandable programs.

3.4 Parallel Processes

Many readers will probably wonder why we talk about parallel processes in a paper that concentrates on scientific software. We argue that parallel processes should be used as an elementary means of algorithmic abstraction in programming. To illustrate this, let us look at a practical problem that has its roots in business-oriented programming but serves our purposes very well:

A publisher keeps a mailing file to produce lables for a periodical, one label per subscriber. Every record holds the subscriber's address and the number of copies he gets. Our program has to print mailing labels, four abreast, to be processed by a labelling machine (e.g. Cheshire). The periodicals must be bundled up as prescribed by the postal authorities:

- If there are 5 or more copies for a single ZIP code (or postal code as it is called in many contries) they have to be collected in one or more bundles addressed to this ZIP code.
- If there are 4 or less copies for a single ZIP code they have to be collected in one or more "compound bundles" addressed to the associated regional distribution office (addressed by, say, the first two digits of the ZIP code).
- No bundle may consist of more than 18 copies. If there are more remaining to be bundled up, collect 15 into one bundle and start a new bundle with the rest.
- Every bundle must be topped by a cover sheet with the full ZIP code or the regional distribution office code if it is a compound bundle. Our program has to produce these cover sheets also. To indicate the end of a bundle to the labelling machine, the program has to augment the last address label per bundle with some optically recognizable sign at a given position.

The data flow in our program will be:

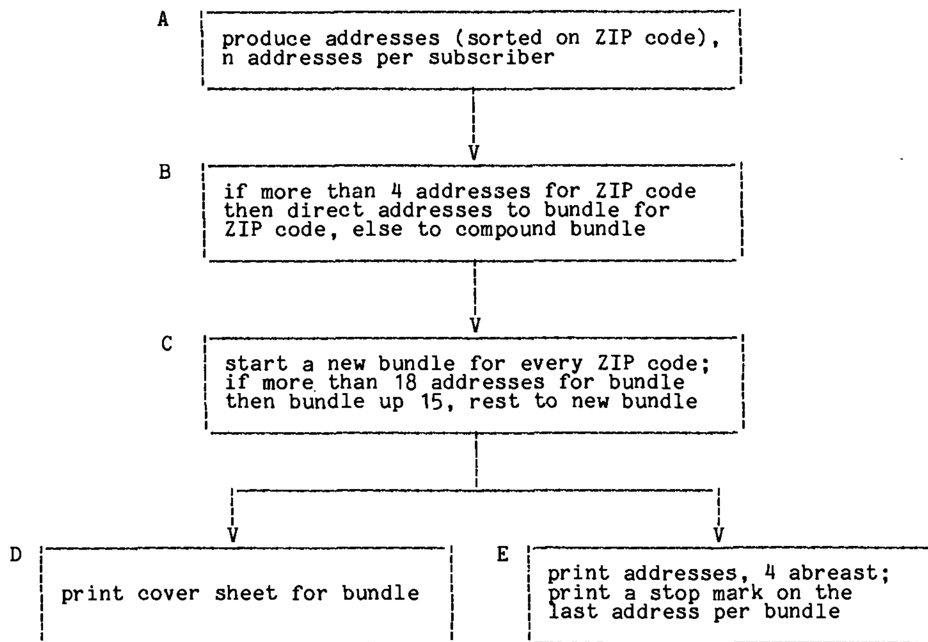


Fig. 1

None of the five modules A to E can be called complex. Modules B and C have to implement a lookahead to take the correct decision, and module E has to suspend printing of the addresses until it has four ready to print. These problems, however, can be easily solved with some temporary storage of addresses. The addresses to be collected into the compound bundle may be written into a workfile by module B. Upon control break on the first two digits of the ZIP code, this workfile is reread and its addresses passed on to module C with the ZIP code of the regional distribution office to which the compound bundle should be mailed.

The most natural way of programming the data flow of Fig. 1 is to create a task (or process) per module, all tasks running in parallel. The tasks communicate with each other to exchange data (addresses in our example). The creation, deletion, and synchronization of tasks as well as intertask communication is implemented by Ada on a very high level of abstraction. Ada's task concept is certainly no more complex to understand and use than other concepts in high level programming languages.

As an example of an Ada task, let us look at the implementation of module C in the data flow of Fig. 1:

```
task MODULE_C is
    entry NEXT_ADDR (A : in ADDRESS);
end;

task body MODULE_C is
    BUFFER  : array (0..4) of ADDRESS;  -- buffer for 5 addresses
begin
    for I in 1..4 loop FILL_BUFFER  -- slots 1 to 4
        accept NEXT_ADDR (A : in ADDRESS) do
            BUFFER(I) := A;
        end;
    end loop FILL_BUFFER;
    while BUFFER(1).ZIP/=9999 loop
        for I in 1..18 loop BUNDLE_UP
            MODULE_E.PRINT_ADDR(BUFFER(1));
            BUFFER(0..3) := BUFFER(1..4);  -- shift left buffer
            accept NEXT_ADDR (A : in ADDRESS) do
                BUFFER(4) := A;
            end;
            exit when BUFFER(0).ZIP/=BUFFER(1).ZIP;
            exit when I=15 and BUFFER(4).ZIP=BUFFER(1).ZIP;
        end loop BUNDLE_UP;
        MODULE_E.PRINT_STOP_MARK;
        MODULE_D.PRINT_COVER(BUFFER(0).ZIP);
    end loop;
    MODULE_D.DONE;  -- signal end to module D
    MODULE_E.DONE;  -- signal end to module E
end MODULE_C;
```

The type ADRESS is assumed to be a record containing an address to be printed. MODULE_C task expects its caller to deliver 4 pseudo addresses with the ZIP code 9999 to signal the end.

As we see, a call to an entry of another task has the same form as a procedure call (e.g. "MODULE_D.PRINT_COVER(BUFFER(0).ZIP)"). An accept statement in a task waits for the corresponding task entry to be called by another task. When called, the statements embraced by "do" and "end" are executed. Waiting for any one of a number of entries to be called is also possible. For a complete explanation of the tasking mechanisms in Ada see reference [2].

Of course, the above problem may be programmed with procedures only, without using tasks. Every procedure would have to remember its internal state in global variables before returning control. The program could even be written without procedures, namely by inverting the data flow into one single program module. Such a program, however, would certainly no longer reflect the structure of the data flow as shown in Fig. 1. Its complexity will be much greater than the sum of the complexities of the single modules A to E.

4 DIVIDE AND CONQUER

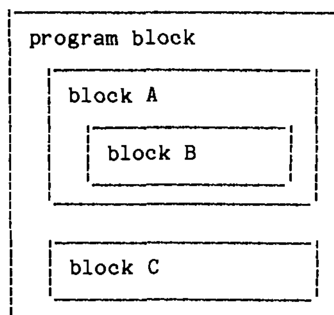
The principle "divide and conquer" is attributed to the Old Romans and kept its importance for the management of all but the simplest systems of any kind. Its message is that a global system should be broken down into subsystems, these in turn again into smaller units, and so on. This division of a system into several subsystems and the associated definition of functional dependencies and intersystem communication is supposed to result in a system that is better manageable than a monolithic system.

Software is nothing else but some kind of a system, albeit not one that is physically perceptible. Hence, "divide and conquer" is also an important guiding principle for software development. It means that a program should be designed as a hierarchical composition of subsystems (or modules) which are easy to program and to maintain. Ada supports the method of "divide and conquer" in software production in many ways. Some of them have already been mentioned in the previous chapters. In addition we shall discuss Ada's concept of block structure, packages, and separate compilation here.

4.1 Block Structure

In FORTRAN, a given identifier is known throughout a program unit (main program or subroutine) but is hidden from other program units (execpt for COMMON identifiers). Such a uniform, global scope of names implies that all statements of a program unit operate on the same name space, i.e. an identifier X always refers to the same object. It is not possible to encapsulate certain data objects in a subunit while still having access to global data objects (information hiding). This is a severe drawback to writing independent program parts in the sense of "divide and conquer" . One has to carefully avoid malicious side effects of functionally independent program parts. Such side effects mainly occur because of modifications to a variable that is expected to remain invariant over the execution of such a program part. The danger of introducing malicious side effects is greatest during program modifications by someone who has not written the original program.

A block structured programming language such as PL/I, Pascal, or Ada eliminates this problem by introducing hierarchically nested scopes of names:



The programmer may declare named objects within every block. A name used in a block refers to the locally declared object with this name. If no such object exists, the name refers to the accordingly named object in the smallest surrounding block. It is never possible to refer to an object declared inside a block from outside this block, but only from the block itself and from all blocks nested directly or indirectly within this block.

Ada associates a separate scope of names with every subprogram, task, and package (plus some other constructs not discussed here). The programmer may also open a scope anywhere in a statement part as follows:

```
    .
    .
    .
    declare
        TEMP:     FLIGHT;
    begin
        TEMP := NEXT;
        NEXT := NEW;
        NEW := TEMP;
    end;
    .
    .
    .
```

Here, the identifier TEMP is introduced as a temporary storage used in swapping the contents of NEXT and NEW which are assumed to be of the type FLIGHT. The scope of TEMP extends to the "end" closing the block.

Block structure is one of the most important attributes of a modern programming language. It supports the method of "divide and conquer" by hiding information to the outside world of a block and by providing a separate name space for every block.

4.2 Packages

In its simplest form, a package represents a pool of common data and type declarations:

```
package WORK_DATA is
    type DAY          is (MON,TUE,WED,THU,FRI,SAT,SUN);
    type HOURS        is delta 0.01 range 0.00 .. 24.0;
    type TIME_TABLE   is array (DAY) of HOURS;

    WORKED            : TIME_TABLE;
    NORMAL_HOURS      : constant TIME_TABLE := (MON..THU => 8.25,
                                FRI => 7.0, SAT|SUN => 0.00);
end WORK_DATA;
```

This package only has a so called visible part. The visible information is given as a sequence of declarations. This information, however, is not directly visible to other parts of the program or, in Ada terminology, to other program units. One way of referring an indentifier declared in a package is by specifying the package name and the identifier in the so called dot notation, e.g.

```
WORK_DATA.WORKED
```

Thus, we might have the assignment

```
WORK_DATA.WORKED(MON) := 5.25;
```

The second form of packages also includes subprograms (procedures, functions) to process the data declared in the package. Assume that we want to include to the package WORK_DATA a function to calculate the overtime and a procedure to update the array WORKED. The visible part of the package would then look like this:

```
package WORK_DATA is
    type DAY        is (MON,TUE,WED,THU,FRI,SAT,SUN);
    type HOURS      is delta 0.01 range 0.00 .. 24.0;
    type TIME_TABLE is array (DAY) of HOURS;

    WORKED          : TIME_TABLE;
    NORMAL_HOURS    : constant TIME_TABLE := (MON..THU => 8.25,
                                 FRI => 7.0, SAT|SUN => 0.00);

    function   OVERTIME (WHEN : in DAY) return HOURS;
    procedure  UPDATE   (WHEN : in DAY; TIME : in HOURS);

end WORK_DATA;
```

This package is obviously incomplete since it lacks the implementation of the two subprograms OVERTIME and UPDATE. They are defined in a separate so called package body:

```
package body WORK_DATA is
    -- internal declarations
    -- internal subprograms
    function OVERTIME (WHEN : in DAY) return HOURS;
        :
        :
    end OVERTIME;

    procedure UPDATE (WHEN : in DAY; TIME : in HOURS);
        :
        :
    end UPDATE;
begin
    -- statements initializing the package
end WORK_DATA;
```

A package body essentially contains the implementation of the subprograms declared in the visible part of the package. Writing these subprograms may in turn require the use of local declarations (variables, constants, types) and local subprograms. They are part of the package body as well, but are not visible outside the package itself, since they do not appear in the visible part. The use of local variables may require the execution of statements to initialize them. These statements are executed when the package comes into existence at run time.

Ada also provides mechanisms to restrict operations on objects declared in the visible part of a package, such that only subprograms of the package itself may operate on these objects. Thus, Ada packages can implement the concept of "abstract data types".

Ada packages are an excellent tool to develop software by the method of "divide and conquer". Unlike simple "include" or "copy" clauses known from other languages, they offer all features necessary to encapsulate data (types, constants, variables) and subprograms providing access to the data. The subdivision of a package into a part visible to the outside world and an invisible part is a very effective safeguard against misuses of the package (intentional and unintentional ones). In some way, the visible part of a package represents a contract between the writer of a package and its user, where the package body is the implementation of the contract.

4.3 Separate Compilation

Ada packages and subprograms may be compiled separately and placed in a library. The speci-

fication of a package (the visible part) and the package body (the implementation of the package) are treated as distinct units and are kept separately by the Ada library mechanism.

A program unit that is compiled may refer to a precompiled package or subprogram. Since not only the implementation of the packages and subroutines but also their interfaces (the visible part of the packages, respectively the definitions of the subprograms) are kept in the library, the Ada compiler can ensure a correct usage of separately compiled subroutines. In other words, Ada supports type-safe separate compilation.

5 PORTABILITY

According to Poole and Waite [3], "Portability is a measure of the ease with which a program can be transferred from one environment to another: If the effort required to move the program is much less than that required to implement it initially, then we say that it is highly portable".

We can expect Ada programs to be more portable than programs written in any other language that is used widely today. Several facts contribute to this property of Ada programs:

- Ada will be the first widely used programming language for which a complete language definition existed before its implementation began. The language definition acts as a coercive standard for all implementations.
- If an implementor calls his language implementation "Ada", it is neither allowed to be a subset of Ada as defined in the Ada Reference Manual, nor to be a superset of it. The latter restriction is a very important rule enhancing program portability: If a certain implementation of a standardized language includes extensions to the standard, the programmers will probably make use of these goodies. The undesirable result are programs written in a "standard language" that are not portable to other implementations of the same "standard language".
- There will be validation software available to check an Ada implementation to be in conformance with the language definition. Any Ada implementation will be checked out thoroughly by the validation software. It may use the name Ada only after it succeeds the validation procedure.
- The tremendous pressure behind Ada (from the world's largest software consumer, the DoD) together with the applicability of Ada in various fields of programming will result in Ada implementations on almost any computer system from micro computers to mainframes.

Even a completely standardized language such as Ada, however, does not guarantee an absolute portability from one environment to another since the interface to certain system software components is very likely to differ between the two. If Ada packages are used to map application oriented interfaces to the actual system interfaces, a high degree of portability can be established. To port the entire software system, these packages would have to be adapted to the new situation while there would be no changes to application programs.

6 CONCLUSION

Ada is a tool for scientific programming which has important advantages over FORTRAN. Its superiority is manifest in all three aspects of software engineering discussed in this paper: In data abstraction issues, in the way a suitable algorithmic abstraction is implemented, and in supporting a software production by the method of "divide and conquer".

A software system written in Ada will not necessarily live up to high expectations, if old-fashioned project management principles and outdated implementation methods are used. It is not difficult to program Ada in a FORTRAN style. There is also a danger of setting up project guidelines which do not take advantage of Ada's features or even hinder their use.

Combined with an adequate project management and modern software engineering principles, however, Ada is a very promising landmark on the way to cheaper software.

REFERENCES

[1] Barnes J.P.G., "An Overview of Ada", Software Practice & Experience, Vol. 10, 851-887, 1980.

[2] "Ada", Springer-Verlag, New York, 1981; Part 1: Ledgard H., "Ada -- An Introduction"; Part 2: U.S. Department of Defense, "Ada Reference Manual".

[3] Poole P.C. and Waite W.M., "Portability and Adaptability", in: Bauer F.L., "Advanced Course on Software Engineering", Lecture Notes in Economics and Mathematical Systems, No. 81, 183-277, Springer-Verlag, Berlin, 1973.

* * *

QUESTIONS

MR. J.T. CARROLL CERN

- Q ---> As a consequence of data abstraction and algorithm abstraction the computer hardware is completely hidden from the user. Is this not in conflict with writing efficient programs.

- A ---> **The execution efficiency is of a lesser concern than development efficiency.**

MR. R. PIKE BELL TELEPHONE LAB. MURRAY HILL, N.J.

- COMMENT ---> The most important thing for an efficient program is the right algorithm.

MR. R. CAILLIAU CERN

- Q ---> Tell something about the support environment of ADA.

- A ---> **The DoD also specifies a programming environment, but the speaker does not want to spend time on this subject.**

MR. R. CAILLIAU CERN

- COMMENT ---> The support environment is 50% of ADA and the way to cheaper software.

MR. J. GAMBLE CERN

- Q ---> Arguments supporting strong typing have been brought forward in several presentations and that in examples types like FRANC, LIRA etc. have been recommended. But then one could not write:
  Variable of type FRANC = Variable of type LIRA*exchange rate

- **A ---> It could be done in ADA, but would be flagged by the compiler.**

MR. H. GROTE CERN

- Q ---> Will ADA programs running on different types of computers with different floating point representations and precision give exactly the same results? (i.e. will they be really fully portable).

- **A ---> For a floating point number declared with *n* digits, full precision will be guaranteed up to the *n*-th digit.**

MR. G. ENDERLE KERNFORSCHUNGSZENTRUM, KARLSRUHE

- Q ---> Are there plans to standardize certain ADA packages? If every implementation created a different package for complex number operations, no portability would exist?

- **A ---> Some packages would have to be standardized.**

MR. G. ENDERLE KERNFORSCHUNGSZENTRUM, KARLSRUHE

- Q ---> How portable are the packages themselves?

- **A ---> Portability will, in first place, be given above the package level. The packages may have to be adapted to different systems and therefore need to be designed very carefully.**

MR. J. ZOLL CERN

- Q ---> Is a standard format foreseen to ensure the portability of data?

- **A ---> This is not a language issue.**

MR.J. ZOLL CERN

- COMMENT ---> The data may be hidden so carefully from the user that he is not able to find them any more.

- A ---> **The writer of an I/O package will know the hardware representation of the data.**

H. RENSHALL CERN

- COMMENT ---> The mass exchange of data will be made very difficult by the many possible data types allowed by ADA.

MR. D. WILLIAMS CERN

- COMMENT ---> The HEP community wants concrete typing, not abstract typing.

- A ---> **Only a small group should be aware of the internal number representation.**

MR. J.G. JEFFERY RUTHERFORD APPLETON LABORATORY

- COMMENT ---> Use either data definitions or subroutine definitions to handle I/O of applications.

- A ---> **This would not be safe and would allow break of the abstraction concepts.**

MR. W. MITAROFF OST. AKADEMIE D. WISSENSCHAFTEN, VIENNA

- COMMENT ---> There have been serious objections recently to ADA by Dijkstra, and the ACM did not recommend ADA to become an ANSI standard, on the grounds that ADA is supposed to be not well defined.

- A ---> **ADA has some ill-defined parts, but the speaker expresses his doubts whether it is possible to define a considerably simpler language suited for software development in the large.**

MR. D. WILLIAMS CERN

- Q ---> When will the statement 'ADA is a better scientific programming language than PASCAL or FORTRAN' become true?

- A ---> **It will be in about 2 years from now.**