Consiglio Nazionale delle Ricerche

ISTITUTO DI ELABORAZIONE
DELLA INFORMAZIONE

PISA

A critical survey of the implementation dependent aspects of Ada

Alessandro Fantechi, Franco Mazzanti

Nota Interna B4-76
Dicembre 1986

A CRITICAL SURVEY OF THE IMPLEMENTATION DEPENDENT ASPECTS OF Ada®

A. Fantechi (CNR-IEI)
F. Mazzanti (CRAI)

Introduction

Although Ada is a standard, the aspects of portability, the predictability of the effects, and the semantic definition of programs making use of some of the features which are "at the border" of the language are very complex and confused. The aim of this work is to clarify the meaning and the effects on the portability of programs of the use of the features of Ada which appear to be "dependent from the implementation", hence lacking a rigorous definition and possibly not implemented.

Note that actually the implementation dependent aspects of Ada are not only those illustrated in chapter 13 of the reference manual (about Representation Clauses and Implementation Dependent Aspects), but many of the "normal" construct can hide some implementation dependent behaviour.

It is NOT issued here the problem of giving a semantics to these aspects, which are often very ambiguously described in the reference manual, and which are still argument of discussion at the level of Language Maintenance Committee/ Language Maintenance Panel meetings.

We are interested instead in trying to put in evidence the different kinds of dependences from an implementation which are allowed by the definition of Ada.

More precisely, section 1 of this paper gives a classification of the implementation dependent aspects, section 2 shows some aspects of the language which can be wrongly seen as implementation dependent, section 3 gives some hints of the effects of a compiler validation policy not fully supported by a rigorous language definition on the portability of programs; section 4 studies more deeply some particular aspect of the language which present implementation dependences, like the input-output system.

In the following, whenever we say "reference manual", we refer to the official ANSI/MIL-STD 1815A Ada reference manual, and references to specific sections of the manual are indicated in parentheses, with the usual dot notation; references of the form AI-00XXX identify the Ada Commentary with the same code, available from AJPO, or by electronic mail from ADA-COMMENT@ADA20.

® Ada is a registered trademark of the U.S. Government (Ada Joint Program Office)

1. A classification of Ada implementation dependent aspects

In this section we attempt a classification of the implementation dependent features of Ada, basing on both their effect on portability of Ada programs and the intrinsic nature of the dependence.
We postpone to section 4 a detailed discussion of some of the features which are here hinted.
In particular in section 1.1 we illustrate the cases in which the legality or the executability of the program can be dependent from the implementation, in section 1.2 we illustrate the cases in which the dependence from the implementation affects only the runtime behaviour of the program, in section 1.3 we illustrate the cases in which the dependences from the implementation might affect the way in which the program interacts with its execution environment.

1.1. Dependencies affecting the legality/executability of an Ada program

Legality and executability of an Ada program may in general be implementation dependent concepts; in particular, several kinds of dependence are possible.
The legality of a program can depend on the predefined environment of the particular implementation, more precisely on implementation dependent definitions which are required to be provided in any case by the reference manual.
E.g.

> A program can make use of the internal structure of the type ADDRESS defined within the predefined package SYSTEM:
>
> ```
> A : constant  ADDRESS := (12333,33473)      -- it might be a pair (segment, offset)
> ```
>
> In this case the legality of a program depends on the structure of the package SYSTEM.
> Notice that some definition for the type ADDRESS is mandatory.

The legality of a program can depend on the use of additional implementation dependent definitions which are allowed by the reference manual.
E.g.

> A program can make use of some additional (implementation defined) numeric types defined within the package STANDARD:

```
Es.     ...                                 -- the definition of the type SHORT_INTEGER
        X: SHORT_INTEGER :=  27283;         -- is neither provided by the user, nor part
        ...                                 -- of the "standard" part of the package
        ...                                 -- STANDARD
```

In this case the legality of a program depends on the structure of the package STANDARD.

A program can make use of some implementation defined attribute (to be specified in the appendix "F" of the manual):

```
M: MASK;
BaseReg : INTEGER := M'BASE_REG;
Display  : INTEGER := M'DISP;
```

In this case the legality of a program depends on the existence of implementation defined attributes. Note that the legality is not affected by the use of implementation defined pragmas, since pragmas not recognized by the compiler have no effect.

A program can make use of the types defined in the package LOW_LEVEL_IO :

```
Dev : LOW_LEVEL_IO.Some_Device_Type := 37;          -- some device information
Inf  : LOW_LEVEL_IO.Some_Data_Type := (561,934894);  -- some data kind
   ....
LOW_LEVEL_IO.SEND_CONTROL (Dev,Inf);                -- some  interaction with the device
```

In this case the legality of a program depends on the structure of the package LOW_LEVEL_IO.
Notice that some package LOW_LEVEL_IO must in any case be provided by the implementation.

In the end the program legality or executability can depend on the semantic restrictions (on the implementation of some features) that an implementation is allowed to impose; these cases are:

If the pragma INTERFACE is not implemented, a main program making use of such a pragma is found to be incomplete (by the linker) and hence refused, even if all the existing (provided) components are perfectly legal.

A program making use of the generic unit UNCHECKED_CONVERSION may get some compile time error during an instantiation of it (depending on the actual types - an implementation is allowed to impose restrictions in this sense).

A program making use of any kind of representation clauses (address clauses, lenght clauses, record and enumeration representation clauses) may become illegal in a different implementation, since it is allowed to restrict their use.

A program can makes use of the types defined in the package MACHINE_CODE:

```
procedure SET_MASK is
  use MACHINE_CODE;
begin
   SI_FORMAT'(CODE => SSM, B => BaseReg, D => Display);
     -- the type SI_FORMAT is defined in the MACHINE_CODE package
end;
```

In this case the legality of a program depends on the structure of the package MACHINE CODE.
Moreover, because the MACHINE_CODE library unit might be not provided at all by an implementation, this case can affect the program legality in even a deeper way (for example a simple context clause mentioning this package might be enough for making the program illegal).

In the end a complete and legal Ada main program may still be not executable if its main procedure has parameters in a form not acceptable following the restrictions that the run-time support may require on them.

1.2. Dependencies affecting the runtime behaviour of the program, due to the definition of the predefined standard environment

In some other cases the dependence from the implementation has less drastic consequences on the portability of the program, in the sense that the original program still remains legal and executable in a different implementation. Almost all the usually considered "fully portable programs" also fall in this class of implemenatation dependent programs because it is very difficult not to depend for example on the range of the predefined INTEGER size. However even if the program results might in principle be affected from this kind of dependences (since the programs in this class are dependent from the implementation from a language definition point of view), the reasonability of the existing implementations and the restrictions imposed by the validation policy (e.g. allowing not to consider implementations in which INTEGERs have a range (-1 .. 1)) greatly restrict the actual cases of program dependence from the implementation (see also sect.3).
More precisely programs falling in this class make implicit or explicit use of predefined values, which are given in the predefined environment.
In particular, such predefined values are:

- the ranges of the predefined numeric types, as defined by the package STANDARD (or elsewhere), visible through the use of the attributes of the numeric types (like FIRST and LAST);
- other values about the implementation of predefined numeric types, returned by other attributes of the types (like MACHINE_EMAX);
- constants defined in the SYSTEM package (like MIN_INT, MAX_INT, etc.)

Where the predefined numeric types interested by the previous observations are:

- the INTEGER and FLOAT types defined in the package STANDARD;
- the DURATION type defined in the package STANDARD;
- the COUNT types defined in the input-output packages (see sect. 4.1).

On implementations in which their use is legal - see previous section - the types to be considered include also the predefined SHORT_INTEGER and LONG_INTEGER, SHORT_FLOAT and LONG_FLOAT defined in the package STANDARD.

Notice that for "use of predefined values" is to intended not only an explicit use for example of the attribute INTEGER'LAST, but also the implicit use of such values, as e.g during the execution of the constraint check during an assignment (X:= 1000000;).

1.3. Dependencies due to interactions with the external world

During its execution, an Ada program interacts with some "external world". The kind of interaction depends on the visibility of the external world that an Ada program in a particular implementation has. An aspect of the external world, which is explicitly mentioned in the reference manual, is constituted by the external files. The interaction between the program and these external files is defined by the input/output operations. This interaction is particularly relevant from the program semantics point of view because it represents in some sense the main "effect" of the program. Though the implementation of the I/O packages is explicitly defined to be implementation dependent, most of their properties appear to be actually related more to some interactions with the external world than to an implementation of the Ada language.

As we will see in section 4, we will find this kind of implementation dependencies not only when we will discuss the Ada input-output system, but also when we discuss other features like time.The execution of delay statements and the evaluation of the function CLOCK indeed implicitly requires some kind of interaction with a real clock which belongs to the external environment.

2. Apparent implementation dependences

In this section three categories of apparent implementation dependences are shown to produce programs whose meaning along the reference manual is given independently from the implementation.

2.1. Freedom in implementative choices

Some aspects explicitly defined as implementation dependent in the manual are not to be considered as such from an abstract point of view. This is the case of the definition of the type FILE_TYPE in the I/O packages or the type TIME in the package CALENDAR, whose explicit definition as "implementation dependent" means that its concrete structure is an implementation choice, but does not affect however the abstract type definition (see sect.4.1). The same can be said for the definition of the predefined numeric operations, whose implementation is said to be "implementation defined" though their abstract properties seem to be fully specified by the language itself.

2.2. Incorrect order dependences

The language definition leaves to the implementation the possibility to choose a particular order in the execution of some activity. In particular the reference manual leaves open this possibility in the following cases:

- assignment statement (5.2) (whether evaluate the name of a variable before to evaluate the expression in an assignment)
- various kind of expressions (3.5, 4.3.2, 4.3.1, 3.2.1, 4.5, 3.6, 6.4, 4.1.2) (order of evaluation)
- library units list (10.5) (order of elaboration)
- component subtype indication (3.6) (order of evaluation)
- parameters in a subprogram call (6.4) (order of evaluation)

Moreover further cases have recently been added by the Language Maintenance Committee/Language Maintenance Panel (see AI-00284)

This linguistic choice is maybe due to the intention of encouraging a programming style which produces more understandable and maintainable programs, discouraging the use of constructs which, though easy to implement, produce less understandable and modifiable programs.

Besides to not impose any order, the reference manual explicitly defines as incorrect those constructs whose effects depend on the order in which their parts might be executed. The presence of an incorrect construct is defined in the manual as a programming error, but with the limitation that a compiler is not required to find out errors of this type (but it is allowed to generate the PROGRAM_ERROR exception if and when such errors are recognized).

In conclusion even if the programmer has the knowledge of some implementation details, say about the order of evaluation of parameters in a subprogram call, is not authorized to use them, otherwise the program is not "implementation dependent" but simply wrong.

2.3. Erroneous executions

The language definition allows an implementation to raise PROGRAM_ERROR when an erroneous execution is started. The fact that the exception is generated requires the presence of quite complex tools of static and dynamic analysis of the program. The situation is similar to the possibility of the generation of STORAGE_ERROR exception (see sect. 4.2), which is partly dependent on the presence of a garbage collector. Hence the language definition cannot impose the presence of such complex tools which migth have an influence on the behaviour of the program.

However a program that can potentially generate PROGRAM_ERROR is not to be considered as implementation dependent, but simply wrong. (From this point of view the situation is different from the STORAGE_ERROR one).

In particular the reference manual explicitly mentions the possibility of starting an erroneous execution in the following cases:

a) due to the use of an undefined value (3.2.1(8), 6.4.1(8))
b) due to changing of a discriminant value (5.2(4), 6.2(10))
c) due to a dependence on parameter passing mechanism (6.2(7))
d) due to an access to a shared variable (9.11(6))
e) due to suppression of an exception check (11.7(18))
f) due to an access to a deallocated object (13.10.1(6))
g) due to an unchecked conversion (13.10.2(3))
h) due to overlay of entities (13.5 (8))

however in some of these cases (eg. cases b,c,d,e,h) the wording of the manual are so ambiguous or contradictory that it is almost impossible to derive any kind of reasonable interpretation of them, nor to understand the original intentions of the language designers.

3. LRM Definition vs. ACVC Validation

The Ada definition allows the implementations to implement in a restricted way (or not to implement at all) some of the language features.

In section 1.1, when dealing with those aspects which may affect the program legality or executability, we have already seen many of these optional features, whose absence in an implementation does not prevent at all its process of validation (e.g. the pragma INTERFACE, the subprogram UNCHECKED_CONVERSION, the existence of a LONG_INTEGER predefined numeric type, most of representation clauses, ...).

However for some other language features (about which the language definition is generally rather vague)

implementation-dependent limitations should be justified; citing the wordings of the summary of AI-00325 (now formally approved as ANSI standard): "An implementation limitation is justified if it is impossible or impratical to remove it, given an implementation's execution environment."

The rationale about that is explained in the response of AI-00325 which remarks that there are many ways an implementation can refuse to support some construct either by refusing to accept the construct at compile time or by raising an exception at run-time. It is recognized here that a technical reading of the manual indeed would allow an implementation to reject almost any program while maintaining conformance to the standard. Why this has happened is explained later: " The Standard is permissive in this respect because specification of acceptable limits was felt to be undesirable, even if possible, in the definition of the language. On the other hand, acceptance of an implementators depends on it behaving in a reasonable manner. In a sense every test in the validation suite is a capacity test that implementations are expected to pass. If any implementation fails one of these tests on grounds of capacity limitations or by raising unexpected exceptions such as NUMERIC_ERROR, the failure must be justified on the grounds that successfull passing of the test is impossible or impractical in the test environment."

So the situation is that:

- Which kind of restrictions are (or are not) allowed depends from the fact that the implementor is able to convince the Ada Validation Office or agencies that the passing of some tests is "impractical in the test environment".
- The I.G., ACVC tests, Ada comments, are the only materials which can give to the implementations and the users some idea of which features of the language are to be reasonably (and to which extent) considered safe from the point of view of the program portability.

  However all this material is still material in evolution as the ACVC tests are released with a six months frequence and Ada comments evolve as new issues are submitted, discussed, and approved.

The conclusion is that, even if the language definition of the reference manual does not guarantee many program properties, a language user can, to some extent, rely on some additional properties of the language which increase its confidence on the portability of the program.

The main problem is that now it is not possible to have a complete picture of this kind of properties, being this issue still in discussion and in continue evolution.

To give just some examples, we can observe that any Ada program smaller than several ACVC test will probably not be rejected by any validated implementation, because its size does not exceeds the implementation capacity tested by the validation; analogously we can rely on the fact that the implementation does not raise STORAGE_ERROR at every subprogram call, or NUMERIC_ERROR at any arithmetic operation, or USE_ERROR at every input/output operation.

Other properties of the language not related to any implementation dependent limitation can be satisfied by most implementation and are enforced by the validation mechanism: e.g. the fact that if a sequential program creates a temporary file, writes some values in it, and then reads some values from the same file still obtains the same values it had written. Also these are important properties from the user point of view, which, even if not required by the

language definition, are in practice required from every validated implementation.

4. Case studies

In this section we study in more detail some specific cases, which present different aspects falling in different classes of those presented previously.

4.1. Input-Output

The I/O system of Ada is centered around the concept of file. More precisely, the reference manual clearly distinguishes between the file (object) - that is, an object of type FILE_TYPE - and the external file - that is, an entity which is external to the scope of the language and which represents the physical file on which data are actually written or read.

From the specification of the operations that can be performed on file objects, as are informally described in sections 14.2.1 and 14.2.4, it turns out that the abstract properties of type FILE_TYPE are completely defined: these properties refer to the status of the Ada file (whether open or closed, and the value of the index) and to the identification of the external file (by its NAME, etc.).

The fact that in the manual FILE_TYPE is defined, in the private part of the I/O packages, as "implementation dependent", means only that an implementation can choose any structure to record the information needed by the allowable operation - and the minimal information needed is deducible from the manual and is not implementation dependent -plus any other information that is used internally by the particular implementation. This is a case falling in the class of apparent implementation dependencies studied in section 2.1.

The external file is an entity whose semantics is completely outside the scope of the language, and implementation-dependent, but on which some operations are vaguely defined in the manual. The effects of such operations can be seen as implementation dependent, in the sence of dependence due to the interaction with the external world (see sect. 1.3). Let us now give some examples showing the vagueness of the definition of the properties of the external file, given in the reference manual.

Paragraph 14.1(1) says: "An external file can be anything external to the program that can produce a value to be read or receive a value to be written". Notice that this definition can include a random input generator or a null device, in which cases nothing can be said about the input source or the destination.

Neither is said explicitly in the manual whether external files can be physical terminals. Since they produce and receive values, it is reasonable to consider that they can, hence comes the question whether a READ operation can be suspensive (waiting for someone digiting a character on the terminal). But does this suspension extend to the task

that has called the READ, or to the whole program? The answer is surely implementation-dependent .
Let us hence try to see what can be said on files from inside a single program. This can be done looking to the operations that in some way affect the external file. Such operations are more or less all the "File Management" ones (14.2.1). They give however a very vague image of the external file. It is hard, although remaining in the single program case, to axiomatize the operations on an external file. We can give three typical cases in which the definitions listed before are not sufficient to tell the behaviour of a program using a file:

a) Sequential use of the same external file

The same program opens an external file with OUT mode and writes to it, then it opens the same external file with IN mode and reads its content.

Several behaviours are possible, depending on the behaviour of the underlying file system and on the kind of physical object denoted by the external file (a file, a terminal, a printer, etc...).

Two typical behaviours can be:

i) not to allow to open a file twice - raising USE_ERROR;

ii) to allow both to write and to read.

b) Concurrent use of the same external file

Two tasks of the same program open the same external file in OUT mode concurrently and start to write to it; at the end, they close the file.

Typical behaviours which even limit to consider the I/O procedures as atomic and to not consider special files are again the twice open error condition or any arbitrary interleaving of write operations.

c) Deleting a file

The diction "ceases to exist" referred to an external file on which a DELETE operation has been performed can be considered the strongest assumption made by the reference manual about the external file. This implies that every successive OPEN on an external file with the same name should fail, unless another external file with the same name has been created in the meanwhile.

However, it is not defined what happens in the case of concurrent use of the same external file by two tasks of the same program: if one of them deletes the file while the other is, say, reading from it, the following possible behaviours, dictated by the external environment, are possible for the second task. However, some of them are in conflict with the reference manual definition.

i) the READ raises END_ERROR, considering that there is no more input (but this is a different state of the file from simply being at the end, since now the function IS_OPEN should return FALSE)

ii) the file is actually deleted immediately, but since the READ is implemented by a sequence of reads from an intermediate buffer, the END_ERROR exception is raised only if the end of the buffer is reached; however,

the value of IS_OPEN should be FALSE in the meanwhile;

iii) the file continues to be considered open at the read side until a CLOSE is issued; but this reasonable assumption contradicts the "ceases to exist";

iv) a behaviour of this kind is not possible since the system does not allow concurrent use of the same file; this seems the closest case to the reference manual definition of the DELETE procedure. This fact can raise the doubt that such a definition hides the requirement for a transactional view of the system.

Till now we have spoken about external file for its observability within a single program. However, if we consider an environment in which several Ada programs are running sequentially or concurrently among them, we can usually have interaction among them through the file system.

The reference manual says explicitly only that nothing is said about the external file after the completion of the program (14.1(7)). Neither anything is said about the possibility of two concurrent programs, one updating the same file the other program is reading.

The interaction among programs via an external file can raise the same kind of problems raised by concurrent access to the same external file by two tasks, discussed above. However, for a particular environment, the relations between the file system and concurrency can be different at the level of task and at the level of programs, hence complicating much more the picture.

In all the discussion above, we have spoken about the external file in terms of (few) requirements made by the reference manual and in terms of (many) reasonable assumptions inferred by our experience with real system behaviours. In practice I/O should be considered almost completely implementation dependent from a semantic point of view, while the standard definition covers only the syntactic aspects of the interface, and very few, and sometimes inconsistent, semantic aspects.

It is to be noticed that the ACVC tests do enforce some "reasonable assumptions" on the semantics of the external file, i.e. they verify that, if implemented, files are what can be considered real usual files. Most of the I/O tests are, as one could expect, implementation dependent.

This is another case showing the distance between the reference manual and the validated implementations, discussed in section 3.

Other minor aspects of the I/O packages that turn out to be semantically dependent on the implementation are the syntax and semantics of NAME and FORM, that can still be seen as an interaction with the external environment (sect. 1.3), the range of the COUNT type (see sect. 1.2.), the consistency check of the read value with the type ELEMENT_TYPE which is used to detect the condition under which the DATA_ERROR exception should be raised (note that this check can also be omitted by an implementation for complex types).

4.2. Storage handling

Features for the storage handling appear in the definition of Ada under different aspects:

1) The predefined storage-related attributes 'SIZE, 'STORAGE_SIZE, 'FIRST_BIT,'LAST_BIT, and the predefined constants MEMORY_SIZE and STORAGE_UNIT of the package SYSTEM. The use of this kind of features does not affect neither the program legality nor the program executability, hence they fall in the cases studied in sect.1.2.

   The actual meaning of the returned value is however fully dependent from the implementation since the reference manual description of the intended semantics for these attributes and constants makes use of terms like" number of bits allocated ...", "number of storage units reserved ..." ,"the offset ... of the last bit...", "number of bits per storage unit", "number of available storage units in the configuration", which are not defined elsewhere in the reference manual, and to which the implementation is supposed to give some intuitive meaning.

   Another predefined attribute should be mentioned, the 'ADDRESS attribute, whose evaluation returns a value of the implementation defined ADDRESS type, the use of which might even affect the program legality, since in general the allowed operations on such values are not defined; in the case that the ADDRESS type is defined as a limited private type, such operations do not even include the assignment or equality operators (see sect. 1.1).

2) Representation Clauses: Only the address clauses, making use of some explicit definition of address values, may affect the program legality in a different implementation. In any case any kind of representation clauses may affect the program executability in a different implementation, because this is a kind of feature which an implementation is not required to implement, without affecting in any way the validation process (see sect. 1.1).

3) Unchecked Deallocation: The UNCHECKED_DEALLOCATION is a predefined generic subprogram which any implementation is required to provide. Its use does not affect in any way the program legality or executability being every implementation forced to provide its definition.

4) STORAGE_ERROR Exception:

   The reference manual avoids in practice to specify rigorously when this exception has to be raised (and this lack of specification is intentional), so in principle this exception might be raised at every instant. In practice however, as mentioned in section 3, the validation policy imposes some implicit constraint on validable implementations, indeed it is at least implicitly required that the STORAGE_ERROR exception is not raised during a validation test for some other language aspect. This means that programs not making a very heavy use of storage (at most as heavy as the ACVC test make) should not get this exception. Programs making

use of storage in an heavier way than the ACVC tests do, might have unpredictable troubles when executed in a different implementation. The problem is made even harder by the fact that the language does not define (and the validation does not require) the existence of some kind of garbage collector for the space used by dynamically allocated objects.

The conclusion is that most aspects related to the handling of the storage do not affect directly the program legality or executability. However the effects of the use of the storage related features of Ada are (as probably expected) almost completely implementation dependent. It is not clear which kind of portability could be expected from programs making use of these features, because even if these programs could in some cases be compiled and executed in a different implementation, nothing ensure that they will continue to behave as in the original implementation.

4.3. Priority

Priorities have been included in the definition of Ada to serve primarily as an indication to the implementation of the relative degree of urgency of the activity peformed by several task (as hinted by the note in (9.8(6))). Indeed priorities should not be used for task synchronization, because their effect on the program execution is completely implementation dependent. Notice indeed that no semantic effects are directly associated to priorities by other language features (for example during a selective wait with several open accept alternatives, the priority of the tasks which are at the first position in the corresponding queues need not to be taken into account by the implementation in order the choose which rendezvous to start).

However the implementation is required in some way to take into account the "relative degree of urgency" expressed in the program by the priorities, depending on what the implementation is able to do in order to execute faster the most urgent task.

In particular the commentary AI-00032 (not yet official ANSI) explicitly requires:

"If an implementation supports more than one priority level, or interrupts, then it must also support a preemptive scheduling policy. For example, if an high priority task executes a delay, then it must be scheduled upon expiration of the delay, preempting a currently executing lower priority task if necessary. In particular, an implementation that runs until blocked is not acceptable."

In conclusion priorities should cannot obviously affect the program legality or executability, and in principle should not even affect the program semantics which is often more nondeterministic than what observable by a program execution in an implementation.

However it seems that priorities should actually have a relevant role in an implementation based on a single processor, possibly strongly affecting the program results. This kind of influence of priorities on the program execution are of course fully implementation dependent and can rarely be reproduced on different implementation.

Notice however that the program semantics (intended as the user expectations from the program execution) should not depend on this feature, otherwise a wrong programming attitude has been taken.

4.2. Time

Time related aspects appear in Ada in several places:

- The predefined constant TICK of the package SYSTEM and the definition of the type DURATION in the package STANDARD.
- The definition of the package CALENDAR introducing the function CLOCK and the type TIME.
- Timed Statements (e.g. delay, timed entry calls, delay alternatives in selective waits).

All these aspects cannot affect the program legality or executabilty, even if the related behaviour are surely implementation dependent.

In particular in the case of the predefined TICK and DURATION definition the dependence introduced from the implementation is simply a dependence on some system values (as illustrated in general in section 1.2), while in the case of the function CLOCK and of the timed statements the dependence introduced by these aspects is of a different kind. The execution of a timed statement and the evaluation of the function CLOCK indeed require some kind of interaction with the environment external to the program (some kind of hardware clock) and the effects of these executions or evaluations depends from the actual behavior of this part of the environment which is outside of the program (for example the hardware clock might be reset by the operator during the program execution); in some sence timed statements and CLOCK introduce the same kind of dependence from the implementation as the I/O operations do.

In both cases this features reflect an interaction with a system not defined and which is not part of Ada.

It is not clear currently which kind of contraints are imposed to the implementation by the validation policy (e.g. whether different evaluation of the function CLOCK should return increasing value of TIME), and this issue is now under discussion at the ISO/AJPO agencies (see AI-00195, AI-00196, AI-00201).