Consiglio Nazionale delle Ricerche

ISTITUTO DI ELABORAZIONE DELLA INFORMAZIONE

PISA

The risk of destructive run-time errors

Piergiorgio Delrio, Franco Mazzanti

Nota Interna B4-31
Luglio 1990

The risk of destructive run-time errors

Piergiorgio Delrio *, Franco Mazzanti **

* Dipartimento di Informatica, Università di Pisa

** Istituto di Elaborazione dell'Informazione,
via S.Maria 46, 56126 PISA, Italy
E-mail: mazzanti@icnucevm.cnuce.cnr.it

Abstract

We introduce the class of destructive run-time errors as those errors after which the execution status of the program can be so damaged that it is usually impossible to perform any kind of program-driven recovery. Programming errors causing this destructive events are analysed, with particular attention to those cases not related to the use of explicitly "dangerous" low level constructs. It is shown how the risk of these errors can be easily underestimated.

1 Introduction

One of the innovative aspects of Ada is the provision of an exception mechanism, which allows the detection of run-time errors. Above all, this mechanism constitutes the base on which some kind of recovery can be planned and performed by the program itself, allowing the development of more robust and safe software (e.g. introducing executable annotations [1] in the source code, and performing appropriate recovery actions when some inconsistency occurs).

However, there is a class of errors which we will call "destructive errors" for which the above scheme does not work. In fact, the occurrence of a destructive error can damage some system structure (data descriptors, run-time system structures, program code) rendering actually impossible the continuation of the program execution.

Errors of this class are obviously far more severe and dangerous than the standard run-time errors manifested by the raising of an exception, especially since in this case there cannot be any kind of recovery plan to be followed by the malfunctioning program.

This kind of error is often identified with the so-called "erroneous executions". However, as we will see later, not all cases of erroneous executions result in destructive errors, and not all occurrences of destructive errors are qualified in the current standard as erroneous executions.

2 The sources of destructive errors

Language implementation (compiler, run-time system) bugs, operating system bugs, and hardware troubles are possible sources for this kind of destructive behaviour. However, in this paper we are we are not concerned with this kind of events.

Programming errors are the other reason for a potentially destructive program behaviour.

First of all we should distinguish between simple logical program design errors (the program is simply wrong, i.e. its semantics is well defined, predictable, correct, but functions differently from the intention of the user), and another category of programming errors related to the incorrect use of some dangerous, not well defined, unsafe, highly implementation dependent or unpredictable Ada features.

Errors of the first type may be, for example, the unintentional deletion of an external file, or more commonly, the wrong update of any user data involving the loss of some critical information. Of course we are not interested in discussing this kind of program design errors.

Errors of the second type are may be related, for example, to the use of some low level features of Ada (e.g. machine code insertions), to the generation of an erroneous execution (e.g. an unchecked conversion between objects of different size), to the use of a not

well defined, or completely implementation dependent aspect of the language (e.g. concurrent I/O operations on the same file object).

All the errors of this second type have the common feature that, in the end, their occurrence might result in the generation of an illegal address, used for a jump or for a memory access.

In the following sections we will discuss in more detail the classes in which this kind of errors can be grouped. Our interest in this particular kind of error derives from the desire to understand better some of the obscure points of the language definition, with the indirect aim of contributing to the reduction of the risk in the usage of the language, providing some input useful, for instance, for a better estimation of compiler selection criteria, techniques for compiler implementations, safe programming styles, and coding standards.

3 Use of low level aspects of Ada

Sometimes the programmer can be assumed to be conscious of the possibility of the occurrence of a destructive error directly caused by the use of some "low level" language feature. In some cases, in fact, the risk in the use of some language aspects is rather explicit ; essentially this is the case in:

- *the use of pragma SUPPRESS* to inhibit the production of code for the execution of standard run time checks.

If an error situation arises and the corresponding check (e.g. a constraint check) is omitted, some memory can be overwritten (e.g. when updating an array component).

- *the use of address clauses* .

If an overlay between an Ada object and a system structure is achieved, the latter may be irrecoverably damaged.

- *the use of machine code.*

Anything may happen if machine codes are not used safely.

- *the use of pragma INTERFACE.*

Note that, as far as we are concerned, an interface to assembler is not different from using machine code directly.

Since the risk in the use of these features is rather explicit we are not particularly interested in further analysing these language aspects.

Two other language features, unchecked conversions and unchecked deallocations, are also associated with explicitly dangerous operations; this fact is properly put in evidence by the reference manual in declaring these subprograms in chapter 13 (Implementation dependent aspects) and giving them expressive names.

While unchecked conversions do not have the immediate effect of generating and using an illegal address, but, at most result in the production of illegal values and structures, the consequences of the wrong use of unchecked deallocations can be more difficult to detect.

The problem with unchecked deallocations is that they have complex and difficult to control side effects on other objects of the program, and on other high level constructs of the language.

Let us consider the following example:

```
type ARRAY_TYPE is array (1..100) of INTEGER;
type REF_TYPE is access ARRAY_TYPE;
REF1, REF2: REF_TYPE := new ARRAY_TYPE;
procedure FREE is
  new UNCHECKED_DEALLOCATION(
                ARRAY_TYPE, REF_TYPE);
...
REF2:= REF1;
FREE(REF1);
```

In this case the execution of FREE(REF1) has the side effect of rendering erroneous (and according to our approach destructive) any use of the value of REF2 as a prefix for a **.all** selection.

However, the relation between REF1 (the object on which the unchecked deallocation is performed) and REF2 (the object that will contain an illegal access value as a side effect of the unchecked deallocation), in this case rather evident, is in general very difficult (if not impossible) to detect.

The situation is made even more complicated in the

presence of object renaming declarations, as illustrated below:

```
INT1: INTEGER renames REF1.all(100);
FREE(REF1);
```

In this case, the side effect of the unchecked deallocation performed on REF1 extends to the definition of the INTEGER variable INT1, rendering erroneous any attempt to use it. A similar situation arises in presence of subprogram calls:

```
procedure DO_SOMETHING (
                VECTOR: in out ARRAY_TYPE;
                COMPONENT: in out INTEGER);
...
DO_SOMETHING(REF1.all, REF1.all(100) );
```

In this case, if FREE(REF1) is executed during the execution of DO_SOMETHING (e.g FREE is indirectly called by DO_SOMETHING, or concurrently called by some other task), any attempt to use the parameters VECTOR and COMPONENT should be considered a potentially destructive error (if the parameters are passed by reference any use of them is a destructive error, while if parameters are passed by copy, a destructive error occurs at the time of the copy-back).

Further cases of complex side effects in the presence of aliasing are possible, in a way very similar to the inconsistent update of an unconstrained record variable (see section 5.4).

4 "Potentially unpredictable" or extremely "lazy" points of the reference manual

Sometimes the language definition is not explicit in requiring the safety (predictability) of some language feature, besides not being explicit in stating its unsafety. As a consequence, there may be a lack of certainty about the possibility of occurrence of destructive errors. This uncertainty, however, will hopefully be reduced when new Ada Commentaries pointing out these unclear language aspects are submitted to AJPO, discussed, approved and transformed into official additions to the language definition.

The result of the above process may be, for example, introducing new cases of potentially unpredictable executions into the language definition (as erroneous executions), or adding new tests in the ACVC validation suite, thus improving the safety of the validated implementations.

Currently, most of these commentaries have not reached a final official status, but are still subject to discussion, revision, and update. In the following we will present a short list of language aspects for which an official position has not yet been assumed, but which we believe may give an interesting contribution to our survey of the possible sources of destructive errors, pointing out the instability of the current standard.

For example, the current trend in the discussion of AI-00867 seems to be in the direction of requiring that attempts to interact with a task after its master program unit is terminated should be considered erroneous. Fortunately this kind of interaction is very unlikely to occur in practice (in meaningful applications); indeed this possibility is considered a genuine "anomaly" in the language definition.

An important clarification comes from AI-00837, which is going to fix a small oversight in the language definition, specifying as erroneous the evaluation of any variable (not only a scalar one) which has become undefined because a task was aborted while updating it.

Another important commentary is AI-00591, discussing the safety of concurrent I/O operations performed by several tasks in parallel on the same file object. A possible consequence of this kind of parallel interaction, in fact, is not only the risk of generating "scrambled" input or output, but also the possibility of unrecoverably damaging some file descriptor or some internal descriptor used by the I/O system or by the

run-time system.
Another relevant language aspect, on which no tentative position has been taken so far , is the safety of the execution of abort statements and of task scheduling (see AI-00447). In some validated implementations, for example, it is plainly stated that the program execution becomes erroneous if a task is aborted while performing some I/O operation. If an implementation is allowed to introduce new cases of erroneous execution in consequence of the use of abort statements, there is the danger that the program execution will be considered erroneous also in the case in which a task is aborted while evaluating an allocator, or while aborting other tasks, or executing any run-time system kernel routine.
Further issues are present in the AI-00xxx list (containing nearly 900 issues) which might have an impact on the possibility of destructive errors in Ada programs; however, it is beyond the scope of this paper to present and discuss them in a more detailed or exhaustive way (see [6], [7], [8],[9]).
Finally, sometimes the language definition is clear, but still leaves an undesiderable freedom to language implementations (for example not imposing any restriction on the possible effects of an erroneous execution, also in the case of a simple dependence on the parameter passing mechanism). The Uniformity Rapporteur Group (URG), an ISO institution devoted to the definition of language recommendations aimed at the reduction of disuniformities among validated language implementations, is going to play an important role in approaching this side of the problem. The degree of safety of those implementations which, beyond being validated, will also take into account the URG recommendations, can be expected to be highly improved (e.g. see the discussion on UI-0018 in section 5.1).

5 High level features with intrinsic but implicit risk

Lastly we come to a third class of potentially destructive errors, which we believe it is extremely important to analyse in detail. There are some cases, in which we cannot in general assume that the programmer is conscious of the danger of the occurrence of a destructive error, since this risk is implicit in the use of high level constructs.
Let us consider, for example, the use of a scalar variable which is supposed to be initialized but is not. As illustrated in Figure 1, and discussed in [UI-0018], [2] [3] the use of such a scalar value as an index selecting an array component used in the left-hand side of an assignment might actually result in some memory overwrite. However, unless very strict safe programming rules are adopted (e.g.see [4]), it is very difficult to ensure that no undefined values will be present in the program.
The situation is even more difficult in the case of real-time embedded programs in which, for example, performance requirements do not permit initializing all of the data structures at declaration time, tasking cannot be avoided (e.g. in order to catch external interrupts), exceptions and interfaces to external devices are widely used, and no garbage collection can be assumed.
In such situations the programmer may be aware of the possibility that the program contains some errors (e.g. it could make use of an undefined scalar value). However this awareness is not localized on a particular "explicitly dangerous" construct to be held under control.
Even worse, the programmer may be only partially conscious of the possible consequences of these errors. For example, it is not at all evident that when an undefined value is used as an array index, the range check on the index value might not be performed at all (because of some dangerous optimizations the implementations are allowed to perform), with the consequence that an illegal address may be generated and that some arbitrary memory location may in the end be overwritten.
This is exactly the set of cases we are going to discuss; i.e. those cases of destructive errors not particularly

related to the use of explicitly "dangerous" low level constructs, and for which the programmer may underestimate the level of risk.

```
subtype SMALL_INT is INTEGER range 1..10;
SMALL_VECT: array (SMALL_INT) of INTEGER;
SMALL_INDEX: SMALL_INT;
-- SMALL_INDEX may become illegal
--
SMALL_VECT(SMALL_INDEX) :=0;
-- The index check is usually optimized by the compiler
-- (the subtype of the expression is exactly the subtype
-- of the index range)
-- If SMALL_INDEX has an illegal value an illegal
-- address is computed and an arbitrary storage
-- location is overwritten.
```

Fig. 1 : Use of an illegal array index generating a destructive error.

5.1 Use of illegal scalar values

In the following we will use the term "illegal value" of a scalar type to denote a scalar value for which there exists no legal literal of the same type with a corresponding bit configuration.

This kind of illegal value can be introduced into the program in several ways (discussed in section 6), the most evident of which is leaving a scalar variable uninitialized.

It is not true that any use of an illegal scalar value should be necessarily considered a potentially destructive error. For example, even if the execution is defined as erroneous, nothing strange should happen after the execution of an assignment Y:=X (where both variables are scalar variables), except for the possible propagation of an illegal value from X to Y.

In [UI-00018] the concept of "critical contexts" is introduced, referring to those contexts in which the evaluation of an illegal scalar value results in the generation of an illegal address subsequently used for a jump or for an update operation. The purpose of this UI is simply to increase the safety and uniformity of implementations by recommending not to optimize the run-time checks needed to detect the presence of illegal values in these "critical contexts".

From a certain point of view, in fact, the use of an illegal address for a read operation is not effectively destructive. Note, however, that depending on the implementation, such an operation might result in an abnormal program completion if some operating system memory protection scheme is violated, still overcoming any attempt to recover this kind of error from inside the program itself. It seems therefore reasonable to consider as cases of potentially destructive errors also those uses of an illegal scalar value which might result in the generation of an illegal memory address for reading purposes.

In some cases a scalar value is used directly to compute an illegal address.

This happens, in particular, when the scalar value is used:

a) as an array index (or as a bound of slice)

b) as index of an entry family

c) as the expression defining the alternative of a case statement (if the implemantation of the case statement makes use of a jump table)

In cases a) and b) an implementation might not detect the illegality of the scalar value, when the subtype of the expression used as index is exactly the same as that used as the array index_subtype. In these cases, in fact, an implementation might decide to optimize the standard range check on the index value, relying on the fact that, in all the cases in which program execution is not erroneous, the check would be redundant (see Figure 1).

In case c), instead, no run-time check is required by the standard on the legality (in the sense specified above) of the value used for the case statement. The program execution has probably already become erroneous if the value of the expression results to be illegal with respect to its type.

In two other cases the evaluation of an illegal scalar value might result not necessarily in the direct and immediate generation of an illegal address, but in the

generation of an inconsistent program status indirectly generating a destructive error. This happens, in particular, when the scalar value is used:

d) during the evaluation of a range definition

e) during the evaluation of a record discriminant.

As illustrated in Figure 2, in case d) the evaluation of an illegal scalar value during the evaluation of the range constraint of a subtype declaration might result in the definition of an illegal subtype which, when subsequently used as the range of a slice, might generate the same destructive effects discussed in point a) above.

```
subtype SMALL_INT is INTEGER range 1..10;
SMALL_VECT: array (SMALL_INT) of INTEGER;
SMALL_INDEX: SMALL_INT;
-- SMALL_INDEX may become illegal
--
subtype SMALL_RANGE is
        SMALL_INT range 1 .. SMALL_INDEX;
-- The subtype check is usually optimized by the
-- compiler the subtype of the range bound is exactly
-- the subtype of the range bound type)
--
SMALL_VECT(SMALL_RANGE) := (others => 0);
-- The index check for the slice bounds can be
-- optimized by the compiler
-- If SMALL_INDEX has a (positive) illegal value an
-- illegal address is computed and an arbitrary storage
-- location is overwritten.
```

Fig. 2 : Definition and use of an illegal range generating a destructive error.

Similarly, the evaluation of an illegal scalar value as a discriminant value, during the elaboration of a subtype declaration or during the creation of a record object, can be considered to produce an inconsistent status easily giving birth to destructive effects during the creation of the object or during its subsequent use. Figure 3 illustrates the case of the creation of a record object with an illegal discriminant, in which the discriminant is used as a subtype constraint for a record component, resulting in the evaluation of an inconsistent aggregate and in an inconsistent initialization of the record object.

```
subtype SMALL_INT is INTEGER range 1..10;
SMALL_INDEX: SMALL_INT;
-- SMALL_INDEX may become illegal
--
type VECT_TYPE is
     array SMALL_INT range <> of INTEGER;
type STRUCT_TYPE (UPPER_B: SMALL_INT) is
  record
    VECTOR: VECT_TYPE( 1.. UPPER_B)
                          := (others => 0);
  end record;
--
MY_STRUCTURE: STRUCT_TYPE (SMALL_INT);
-- The subtype check on the discriminant value can be
-- optimized by the compiler
-- If SMALL_INDEX has an illegal value an illegal
-- range is computed and a sequence of arbitrary
-- storage locations is overwritten.
```

Fig. 3 : Definition and use of an illegal discriminant as subcomponent subtype constraint.

Figure 4 illustrates the case of the creation of a record object with an illegal discriminant, in which the discriminant itself is used as value for a record variant part, resulting in the creation of an object with an inconsistent structure. In this case it is not guaranteed that the object creation can proceed without generating inconsistencies in the program status.

```
subtype SMALL_INT is INTEGER range 1..10;
SMALL_INDEX: SMALL_INT;
-- SMALL_INDEX may become illegal
--
type STRUCT_TYPE (TAG: SMALL_INT) is record
  case TAG is
    when 1..5  => COMP1: INTEGER;
    when 6..10 => COMP2: BOOLEAN;
  end case;
end record;
--
MY_STRUCTURE: STRUCT_TYPE (SMALL_INT);
-- The subtype check on the discriminant value can be
-- optimized by the compiler
-- If SMALL_INDEX has an illegal value an illegal
-- case variant is evaluated.
-- The object has a completely unpredictable structure.
```

Fig. 4 : Definition and use of an illegal discriminant as record variant.

Note that if an implementation does not optimize also the subtype checks associated to type conversions, it is sufficient to include all expressions in the contexts a) .. e) above in an explicit type conversion, to resolve the problem at run-time detecting all dangerous uses

of illegal scalar values.

For tha sake of completeness it is interesting to observe the existence of contexts "less critical" than those of cases a) ... e) above, in which the use of an illegal scalar value does not result in a destructive behaviour (neither directly nor indirectly), but can still originate a very "anomalous" program behaviour. The use of illegal boolean values as conditions for loops and if statements (In Figure 5 it is illustrated the case of an endless loop), provides a typical examples for this kind of anomalous behaviour.

```
COND: BOOLEAN;
EXITING : BOOLEAN := FALSE;
-- COND may become illegal
--
loop
   if (COND = TRUE) and EXITING then ... ; exit;
   end if;
   if (COND = FALSE) and EXITING then ... ; exit;
   end if;
   -- initially at least one cycle is performed
   -- (EXITING = FALSE)
   EXITING := TRUE;
   -- if COND contain an illegal value, even if
   -- EXITING becomes TRUE,
   -- none of the above two exit conditions might
   -- evaluate to true.
end loop;
```

Fig. 5 : An endless loop.

5.2 Use of illegal access values

Intuitively, an access value can be considered illegal when it does not denote a currently accessible object created by the execution of the standard allocator inside the access type collection.

Illegal access values cannot be introduced into the program by uninitialized variables, but can still be generated as a consequence of other cases of erroneous execution or destructive errors (see sect. 6).

Although not all uses of illegal access values necessarily constitute a potentially destructive error (consider for example the execution of a simple assignment propagating an illegal access value from one access variable to another), the association of illegal access values with illegal memory addresses is rather immediate, and any evaluation (implicit or explicit) of the **.all** selector applied to an illegal access value can be considered a case of destructive error.

We can observe that, in this case, unlike in the case of illegal scalar values, there is no run-time check the programmer can perform to detect the illegality of the access value. It is therefore particularly important to avoid the generation of illegal access values, when program safety is a major requirement.

5.3 Use of illegal record structures

Record objects with discriminants constitute a very powerful language feature, which however may give rise to severe problems.

A record structure is considered to be illegal in two cases:

a) Some of the record discriminants are illegal.

b) All the record discriminants are legal scalar values, but the remaining record components do not reflect the structure required by the discriminant values.

In both cases there is a basic uncertainty even on the simple existence of the record components which depend on the discriminant.

We have already said in section 5.1 that the attempt to create of a record object with an initial illegal structure (i.e. evaluating an initial illegal discriminant value) should be considered a potentially destructive error.

The situations is not different if we consider the possible effects of the use of a record object which has become illegal (i.e. it has got an inconsistent structure) during the program execution.

As in the case of access values, these inconsistent record structures cannot be introduced into the program by uninitialized variables, but can still be generated in consequence of other cases of erroneous execution or destructive errors (discussed in sect. 6) .

As well as for the case of access values, it is in general impossible to detect at run-time the illegality of the structure. On the contrary, there is no use of these illegal structures which can in general be considered safe.

The danger of using an inconsistent record structure

becomes apparent if we consider the possible implementation choice of using hidden pointers for allocating in the heap the memory used for the record subcomponents corresponding to the record variant parts or the record subcomponents depending from the discriminant itself.

Once an inconsistent record structure is produced, any use of the record itself may result in the evaluation of an illegal memory address. It is therefore particularly important, also in this case, to avoid the generation of illegal structures when program safety is a major requirement.

5.4 Non-atomic access to unconstrained record objects and components

Unconstrained record variables constitute another high level but "dangerous" aspect of the language.

The problem with this kind of objects is that their internal structure changes according to the current value of the discriminant, but their update is not required to be atomic.

Consequently, when the use of an unconstrained record (or of its subcomponents) overlaps with an assignment changing the record structure, the result can be a destructive error.

This may happen, typically, when an unconstrained record is shared among several parallel tasks, and accessed in a non synchronized way (e.g. written and read simultaneously). This kind of erroneous behaviour usually results in the generation or evaluation of illegal values, but in this case it may also cause the production and use of an illegal address, thus generating a destructive error.

As already discussed for the case of unchecked deallocations (in section 3), any use of a formal parameter corresponding to an unconstrained record (component) is erroneous and possibly destructive if the variable used as actual parameter is updated during the execution of the subprogram call directly by the subprogram itself or by some other task.

Finally, the update of an unconstrained record as a side effect of a function call can result in the evaluation and use of an illegal address if the evaluation of the function call occurs during the evaluation of the name of a subcomponent of the unconstrained record, or inside an assignment to a subcomponent of the record. As illustrated in Figure 6, this situation is not likely to occur frequently, and it should not be difficult to devise some "safe programming style" avoiding the generation of these situations.

```
type VECT_TYPE is array (1..10) of INTEGER;
type STRUCT_TYPE (TAG: SMALL_INT := 1) is
record
  case TAG is
    when 1..5  => VECTOR: VECT_TYPE;
    when 6..10 => COND: BOOLEAN;
  end case;
end record;
--
MY_STRUCTURE: STRUCT_TYPE;
-- Unconstrained object:  initial discriminant value is 1;
INT_VAR: INTEGER;
--
function BAD_FUNCT is
begin
  MY_STRUCTURE:= (6,FALSE);
  return 10;
end BAD_FUNCT;
--
INT_VAR :=
      MY_STRUCTURE.VECTOR(BAD_FUNCT);
-- When the array prefix is evaluated, the object
-- structure is correct.
-- When the index value is evaluated, the object
-- structure is also changed,  so that, when the selection
-- is performed, an illegal address is used.
```

Fig. 6 : Non atomic access to an unconstrained record component.

6 Generation of scalar illegal values, access illegal values, illegal structures

In the previous section we have pointed out how, given the presence of various kinds of illegal values and structures in the program, several high level usually safe constructs (array indexing, access value dereferencing, assignments) turn into potential sources of destructive errors.

A first important source of illegal values and structures is the set of *low level language aspects* already mentioned in section 3: unchecked conversions, unchecked deallocations, suppression of

exceptions, machine code insertions, interfaces to other languages, address clauses. Most of these features are also directly responsible for destructive errors. We are not interested in discussing these aspects in detail, the risk of their use being explicit.
Another important source of illegal values and structures is obviously the *occurrence of a destructive error*, as illustrated in sections 3, 4 and 5. A destructive error, in fact, instead of damaging some system RTS descriptor, or performing an unrecoverable jump to an illegal instruction, can simply damage some user data. Neither in this case are we interested in a more detailed discussion.
Finally, other *high level constructs* are often responsible for the generation and propagation of illegal values and structures, and these we are going to discuss in more detail in the next subsections.

Lack of object initialization
Ada does not define default initial values to be assigned to objects upon creation, neither is the programmer required to provide them explicitly. Consequently, uninitialized objects may have undefined values. Exceptions to this rule are the case of access types (objects are initialized by default with a null value) and the case of records with discriminants, for which an initial discriminant value must be specified.
All scalar objects, and all scalar subcomponents of record and arrays, are allowed to contain undefined and possibly illegal values.
Once undefined scalar objects are created, their undefined value is easily propagated when they are used in right-hand sides of assignments and in subprogram calls. Officially program execution becomes erroneous when such an undefined value is evaluated, but in practice nothing unpredictable happens, and the undefined value is copied from one object to another.
On the other hand, the evaluation of partially undefined record and array values (i.e. record and array values whose scalar subcomponents are undefined) is not considered erroneous, and the assignment or use of these as actual parameters are considered correct operations. It should be noted, in fact, that there are no "critical contexts" which could be used unsafely with these non-simple values.

Undefined copy-back value of out parameters
A situation very similar to the previous one arises when no (defined) value is assigned to a scalar "**out**" parameter of a subprogram before the completion of the call. In this case, when the subprogram call is completed (without raising an exception), the execution of the copy-back propagates the undefined value from the formal parameter to the actual one. On the other hand, if the subprogram call is abandoned because of an exception, no copy-back is performed and the undefined value is not propagated.
Note that this kind of propagation occurs only in the case of formal parameters (or subcomponents of formal parameters) of scalar type, since, in the case of parameters (or their subcomponents) of access type, the copy-in for the component is performed also if the formal parameter has mode "**out**". Similarly, copy-in is always performed for the discriminants of formal parameters (or their subcomponents) as well.

Not synchronized use of shared variables
The simultaneous access of two tasks to the same shared variable may easily result in the production or evaluation of an illegal scalar or access value, or of an illegal record structure.
In particular, if both tasks attempt to simultaneously update the same variable, the final value stored into the variable may be different from any of the two values which were intended to be assigned, resulting as an interleaving of the two byte sequences.
On the contrary, if one task attempts to read a shared variable (of any type), while another task is updating it, the final value of the variable will certainly be the newly assigned value, however nothing prevents the other task from reading part of the old value and part of the new value, thus cuasing the evaluation of an

illegal value.
In fact, the update and read of a shared variable (scalar, access, record or array) is not required to be atomic, but can be performed as a sequence of read or update operations on a byte by byte basis.
This kind of behaviour is not restricted to parallel architectures, in which several processors simultaneously execute different, and possibly interfering, tasks. On a mono-processor architecture as well we might experience these effects if task scheduling is implemented on a time-slice basis or adopting a preemptive policy.
The impact of this unsafe language aspect on program security is particularly high since it is not always possible to adopt the safe policy of not using any shared variables at all. Moreover, it is not at all easy to prove the absence of unsynchronized accesses to a shared variable in any possible program execution.
Last, but not least, once illegal access values and illegal structures are evaluated or generated, it is practically impossible to detect their illegality.
The pragma SHARED, even if in theory should allow the user to require atomic read/update operations on shared variables, in practice is rather useless being rarely supported by an implementation, and being applicable neither to non-scalar objects nor to subcomponents of scalar objects.

Unchecked input

The Ada packages DIRECT_IO and SEQUENTIAL_IO provide a means for reading any type of data (except for values of a limited private type) from an external file.
An implementation, at least in theory, is required to raise the exception DATA_ERROR if the element read cannot be interpreted as a value of the required type. However, since this check can be omitted if performing it "is too complex", the common practice is not to perform the check even in the case of scalar types.

As a consequence, input/output becomes an open door for illegal values and structures of any type. In practice, input/output does not seem very much different from the other low level features present in the language. The problem here is that this danger is not sufficiently exposed to the attention of the programmer, who tends to imagine that, after having instantiated a generic package on its scalar subtype, the read operations performed are safe operations returning values of the scalar subtype in question, or raising some exception. From this point of view DIRECT_IO/SEQUENTIAL_IO READ operations can be considered low level features well disguised as high level constructs.
A similar situation occurs in the case of GET operations reading values of a CHARACTER or STRING subtype (defined in the package TEXT_IO). Note, in fact, that the representation of CHARACTER requires only 7 bits, while most implementation simply use one full byte for storing a character value. As a consequence, when character GET operations are used to read a binary file, 8 bit sequences are read from the file (or device) thus generating potential illegal caracter values. It is curious to observe that the Reference Manual in this case does not even allow an implementation to raise DATA_ERROR when an illegal character is found.

Abort statements

We have already pointed out in section 4 the possible unsafety of abort statements whose effects are not explicitly required to be safe, for instance when a task is aborted while performing some critical activity on some system structure. The conclusion was that this language aspect was not clear and that it needed an official clarification.
On the contrary it is explicitly specified in the reference manual that update operations are not required to be atomic with respect to abort statement

and that, if a task is aborted while updating a variable, the variable itself becomes undefined.
This means, in particular, that illegal scalar values and illegal access values can be produced as a consequence of an aborted update operation, as well as inconsistent record structures with illegal discriminants.
We can observe that this risk is present only in those implementations actually supporting "immediate aborts", and that this risk is completely avoided by implementing abort statements in a non-immediate way, i.e. labelling the aborted task as "abnormal" and allowing it to continue execution until it reaches a point at which it can be forced to complete safely.

The conclusion of this survey is that abort statements, unchecked read operations, and shared variables are three high level language features which allow the production of illegal access values and illegal record structures. The danger of their use is particularly high since, once this illegal values are generated, their illegality is almost impossible to detect performing any kind of user-defined checks, and their use may result in destructive errors.
Missing initializations and undefined out parameters are additional sources of illegal scalar values. These values as well can result in destructive errors when used in a particularly "critical" context, however their illegality can usually be checked by the programmer performing explicitly the appropriate type checks.
Finally, remember that also the various kinds of side effects caused by wrong unchecked deallocations may produce illegal scalar and access values, and illegal structures. Even if these errors are generated by the wrong use of an explicitly dangerous low-level feature, it may not be easy to understand the association between the use of such an illegal value and a previous use of this low level feature.

7 Destructive errors and erroneous executions.

The Ada concept of "erroneous execution" at a first glance seems to match exactly our definition of "destructive error", at least since the program execution is officially said to become "unpredictable" when an erroneous execution starts.
At a more detailed analysis, however, we can see that several cases of officially "unpredictable" executions in practice behave in a rather predictable and relatively "safe" way.
Let us consider, for example, a dependence on the parameter passing mechanism.
As illustrated in Figure 7, a subprogram whose semantics depends on the parameter passing mechanism (and this is yet another ambiguous point in the LRM definition) may not follow the underlying programming style encouraged by Ada, yet cannot be considered a destructive error.

```
type VECT_TYPE is array (1..10) of INTEGER;
MY_VECT: VECT_TYPE;
--
procedure IMPLEMENTATION_DEPENDENT
                    (VECTOR: in out VECT_TYPE) is
  SOME_EXCEPTION: exception;
begin
  VECTOR:= (others => 0);
  raise SOME_EXCEPTION;
end IMPLEMENTATION_DEPENDENT;
--
IMPLEMENTATION_DEPENDENT (MY_VECT);
-- the effect of the call depends from the parameter
-- passing mechanism selected by the implementation:
-- if copy-back is adopted, MY_VECTOR is not
-- updated, otherwise MY_VECTOR is initialized
-- with all 0's .
```

Fig. 7 : A subprogram whose semantics depends on the parameter passing mechanism.

Figure 8 is a list a "robust" Ada types, in the sense that all the possible bit patterns for type values are legal Ada values. In this case, even in the presence of undefined scalar values we are sure that no illegal values can be produced. The evaluation of such an undefined but legal value in any critical or non-critical context might result in a certain degree of nondeterminism in the program execution, but not in a destructive error.

```
type MY_INTEGER is new INTEGER;
-- Usually there are no illegal values of the
-- predefined INTEGER type;
--
type BYTE is new INTEGER range 0..255;
for BYTE'SIZE use 8;
-- The length clause forces the use of no more than 8
--  bits for the representation of the objects of this type..
-- Under these assumptions, no illegal values of this
-- type can be produced.
--
type SAFE_BOOLEAN is new BOOLEAN;
for SAFE_BOOLEAN'SIZE use 1;
-- Logical operations are more expensive, but TRUE
-- and FALSE are the only possible values.
```

Fig. 8 : A list of "robust" Ada types.

Similarly, the use of an illegal scalar or access value in any non-critical context is considered an erroneous execution by the language definition though it causes, at most, a propagation of the undefined value.
In other cases the language definition requires the program execution to become erroneous while in our approach it only results in the generation of illegal values. This is the case, for example, of unchecked conversions not preserving the properties of the target type, or the case of not synchronized accesses to a shared scalar variable. From our point of view it is only the subsequent use of these values or objects which might actually be unpredictable, and not the activity of producing them.

Finally, the current standard does not qualify as erroneous several cases of unpredictable program executions (e.g. the use of an illegal scalar value obtained from a SEQUENTIAL_IO READ operation, if the DATA_ERROR check is not performed by the implementation, or from a TEXT_IO.GET operation reading characters from a binary file). Sometimes these language omissions are clearly to be considered small Reference Manual oversights; in other cases (e.g. those mentioned in section 4) it is not so clear whether these omissions should be considered oversights or just as reflecting an underlying rationale taking more into consideration program performance or implementation simplicity rather than program safety.

8 References

[1] D.C.Luckham, F.W. von Henke, B.Krieg-Brückner, O.Owe: "Anna — A language for Annotating Ada Programs. Technical Report 84-261, Computer Systems Laboratory, Stanford University, July 1984.

[2] F.Mazzanti: "Reducing unpredictability in Ada executions" ACM Ada Letters, Vol. 9 n.6 pag. 90-96, September 1989.

[3] B.A.Wichmann, "Insecurities in the Ada Programming language: An interim report", NPL Report DICT 137/89, January 1989.

[4] B. Carre, T. Jennings: "SPARK- The spade Ada Kernel", Department of Electronics and Computer Science, University of Southampton. 1987.

[5] R. Holzapfel, G.Winterstein: "Ada in safety critical applications" Proceedings of 1988 Ada-Europe Conference "Ada in Industry", June 1988, also The Ada Companion Series, Cambridge University Press.

[6] S&M: "Erroneous Executions in Ada: the current state of definition", Deliverable 1 of Task 4.6 of EEC MAP project n. 755 "SFD-APSE", December 1988

[7] S&M: "Erroneous Executions in Ada: Towards Run-Time detection", Deliverable 2 of Task 4.6 of EEC MAP project n. 755 "SFD-APSE" , December 1988

[8] F.Mazzanti: "The Point on Ada erroneous executions", IEI Internal Report n. B4-05, March 89.

[9] P. Delrio: "A solution to the problem of erroneous executions in Ada", Graduation Thesis in Scienze dell'Informazione, Dipartimento di Informatica,University of Pisa, April 1990 (in italian).

[AI-0xxxx] Ada Rapporteur Group commentary n. xxxx. Available by anonymous FTP from the host ajpo.sei.cmu.edu <128.237.2.253>, from the directory ada-comment.

[UI-00yyy] Uniformity Rapporteur Group commentary n. yyy. Available upon request from Dr. Brian Wichmann, Building 93, DITC, National Physical Laboratory, Queens Road, Teddington, Middx TW11 OLW, U.K., <ada@seg.npl.co.uk>.