Exposing Uninitialized Variables: Strengthening and Extending Run-Time Checks in Ada

Robert Dewar[1], Olivier Hainque[2], Dirk Craeynest[3], and Philippe Waroquiers[3]

[1] Ada Core Technologies
Fifth Avenue, 73, NY 10003 New York, United States of America
dewar@gnat.com
[2] ACT Europe
Rue de Milan, 8, 75009 Paris, France
hainque@act-europe.fr
[3] Eurocontrol/CFMU, Development Division
Rue de la Fusée, 96, B-1130 Brussels, Belgium
{dirk.craeynest,philippe.waroquiers}@eurocontrol.int

Abstract. Since its inception, a main objective of the Ada language has been to assist in the development of large and robust applications. In addition to that, the language also provides support for building safety-critical applications, e.g. by facilitating validation and verification of such programs. The latest revision of the language has brought some additional improvements in the safety area, such as the Normalize_Scalars pragma, which ensures an automatic initialization of the non-explicitly initialized scalars. This paper presents Initialize_Scalars, an enrichment of the Normalize_Scalars concept, and an extended mode to verify at run-time the validity of scalars, both designed for easy use in existing large applications. Their implementation in GNAT Pro (the GNU Ada 95 compiler) is discussed. The practical results obtained on a large Air Traffic Flow Management application are presented.

1 Introduction

One common cause of bugs that are difficult to find is the use of uninitialized variables. They often lead to unpredictable behaviour of programs, showing up only under special circumstances not necessarily encountered during testing. As an example, one release of the Eurocontrol CFMU[1] Air Traffic Flow Management application [1] had such a bug in compatibility code used to reload the binary data produced by the previous release. When a new field is introduced, the old binary data must be read and a default value must be given for this new field, resulting in code similar to the following:

[1] Eurocontrol is the European organization for the safety of air navigation. CFMU is the Central Flow Management Unit within this organization, in charge of the European flight plan processing and air traffic flow management.

```
if Reading_Current_Version then
   Boolean'Read (Stream, A_Flight.New_Field);
else
   A_Flight.New_Field := True;
end if;
```

This sequence was wrongly coded as follows, leading to an uninitialized New_Field for all the flights computed with the previous release and re-read with the new version:

```
if Reading_Current_Version then
   Boolean'Read (Stream, A_Flight.New_Field);
end if;
```

This bug was not detected during operational evaluation of the new release despite the extensive testing (including multiple binary data migration rehearsals). It was nonetheless triggered during the real migration, resulting in some temporary inaccuracy in the computation of the air traffic control load. The effect was limited to one elapsed day, since the new flights were computed by the new version of the software that correctly initializes this New_Field.

To avoid occurrences of similar bugs in other cases, CFMU deemed it necessary to investigate the practical ways of detecting uninitialized variables in this large application under constant evolution. The main objective was to improve the robustness of the application, preferably by using automated tools and techniques. A major constraint was the performance requirements of the system.

At the beginning of this investigation, Ada Core Technologies and CFMU discussed possible approaches. These discussions resulted in the specification and implementation in the GNAT compiler of features that help to detect uninitialized variables with additional run-time checks.

2 Detecting Uninitialized Variable Usage

Various techniques and methods are already available to track down uninitialized variables. This can be done either at "code level" (before running, using code analysis techniques) or at run-time (by insertion of code whose only purpose is to detect such usage).

2.1 Static Detection of Uninitialized Variables

Formal Validation Techniques: One well-known way to avoid using uninitialized variables is to avoid writing code that creates them. However, such approaches (e.g., formal validation techniques like SPARK [2]) are difficult to apply on large-scale applications like Eurocontrol's, which by their nature make extensive use of various features (e.g. tasking, exceptions, recursion, dynamic memory, usage of COTS products, and so forth). It is even less practical if this approach has to be applied on a large already existing code base (1.5 million Ada source lines for the CFMU application).

Compiler Warnings: In some cases, classical compiler principles can be used to produce warnings about dubious code (e.g. usage of a variable before it is initialized). Typically, the GNAT compiler emits such warnings in various cases, through front and back ends circuits that trace possible static flow paths. All such warnings were corrected in the CFMU application when it was converted from Ada83 to Ada95 [3].

However, the problem of statically detecting all and only the error cases can be shown to be equivalent to the halting problem, and thus has no general solution. In particular, array elements are not easily tracked, and no set of warnings can catch all the cases without generating an excessive number of false positive warnings.

2.2 Detecting Uninitialized Variables at Run Time

Purify-like Solutions: Purify is a well-known tool that helps to detect uninitialized variable usage, amongst other nice features and capabilities. It inserts checks in the compiled object code, which means that having the source code is not necessary. In addition, by maintaining a "shadow" memory to track initializations, Purify can detect invalid uses without requiring invalid bit patterns.

However, this approach has some limitations. The technology is very advanced but complex and does sometimes introduce difficulties, e.g. when transforming COTS libraries. Also, the instrumentation is highly dependent on which compiler is used and in which environment. For example, the following code:

```
main ()
{
  int i;
  i = i + 1;
}
```

could not be handled properly by Purify version 5.2 and 5.3 beta on HP-UX 11 when compiled with aCC (even an instrumented correct program gave a memory fault). When compiled with cc, the transformed code correctly detected the invalid usage of variable `i`. When compiled with the gcc C compiler provided with GNAT 3.15w, Purify correctly detected the invalid usage of `i`. On Windows NT with Microsoft Visual VC++ 6.0, the misuse of `i` was not detected by Purify (but the instrumented code was not giving a segmentation violation).

The above problems probably originate from the fact that Purify works at object code level. This is both a strength (no need for the source) but also a weakness, since it seems difficult to insert in the object code the additional assembly instructions that will detect an access to uninitialized memory.

Purify is also a "all or nothing" tool that cannot be applied selectively: it searches for all problems it can search in all object files and libraries. The resulting instrumented code is significantly larger and slower than the original code (3 to 5 times slower, taking about 40% more memory [4]), which often precludes the usage of 'Purified' applications in an operational context.

In the past, Purify was successfully used in CFMU to detect memory leaks inside Ada code compiled with GNAT. However, it did not detect the usage of uninitialized scalars that were later detected using the Initialize_Scalars GNAT enhancements.

Normalize_Scalars: After initial discussions between ACT and Eurocontrol, one of the suggestions was to use the Normalize_Scalars feature of Ada95 to detect this bug at run-time or to at least obtain a predictable behaviour of the application. However, this pragma was designed for the purpose of eliminating non-determinacy from safety-critical programs, which is why it appears in Annex H of [5]. This is a somewhat different goal from detecting uninitialized variables, and consequently Normalize_Scalars has limitations that make its use difficult for large applications.

Application wide consistency - Normalize_Scalars is a configuration pragma that implies that the full partition is compiled with this pragma, including the Ada run time. This problem can of course be solved by having the compiler vendor supply a pre-compiled run-time for use with Normalize_Scalars. In the case of GNAT, this solution is even simpler as the run-time code is available and can be recompiled easily. In case the application is integrating COTS products for which sources are not available, special arrangements must be made with each of the COTS providers so that a Normalize_Scalars version is provided. In any case, the requirement for partition-wide consistency is quite inconvenient, and precludes the use of Normalize_Scalars for testing small parts of a large application.

Invalid values only if possible - The Normalize_Scalars specification ensures that if possible, an invalid value (out of range) is used to initialize otherwise uninitialized scalar objects. Of course, if the full bit range is covered by the values of the scalars, then there is no invalid value. In this case, Normalize_Scalars ensures a predictable behaviour by initializing to a "normal" value but cannot detect the usage of this "normal" value.

Manual coding to detect invalid values - The Ada95 language revision added features to check the validity of a bit pattern (cf. Ada Reference Manual [5], RM H.1.1.) In conjunction with Normalize_Scalars, it is thus possible to detect errors with code such as:

```
if A_Flight.A_Field'Valid then
  .... -- this field can be used
else
  .... -- error handling
end if;
```

In practice, this kind of manual technique cannot be applied on a large scale application, essentially because of the existing code base size, so an automatic validity check would be more useful.

As the use of uninitialized variables in Ada 95 should not “by itself lead to erroneous or unpredictable execution” ([5], RM 13.9.1-11), a compiler is free, and sometimes required, to insert “hidden” code checking the validity of scalar values. In practice, such constructs as case statements or array assignments may require checks avoiding wild jumps or memory corruption, but most other operations can simply proceed with invalid data, possibly leading to invalid results.

If we want to maximize the chance of discovering usage of uninitialized variables during testing, then it is desirable to increase the amount of checking done at run-time. In case there is no invalid value for the type bit range, an alternative solution has to be found to detect access to uninitialized data.

3 GNAT Enhancements: Initialize_Scalars pragma and New Checking Levels

To overcome the limitations of Normalize_Scalars, CFMU and ACT undertook to specify and implement an alternative approach to Normalize_Scalars, resulting in an enhancement contract to develop:

- a new pragma Initialize_Scalars,
- a way to choose the initial values of otherwise uninitialized scalars,
- compiler support for fine grained additional validity checking levels.

3.1 The pragma Initialize_Scalars

The pragma Initialize_Scalars ensures, as does the Normalize_Scalars, an initialization of otherwise uninitialized scalars. However, the constraint that the pragma must be used for the whole partition has been removed. This means that it can be used in a much wider scope than Normalize_Scalars, whose primary target was embedded safety-critical applications that rarely use COTS libraries. It also makes it convenient to use for a small part of a large application, and avoids the requirement of recompiling the run-time library. (Cf. GNAT Reference Manual [6]).

3.2 Choice of Initial Value

The standard requires Normalize_Scalars to initialize to an invalid value if possible and requires the documentation of cases in which no invalid value can be generated. In order to be able to detect more usage of uninitialized variable, the initial value used by Initialize_Scalars can be chosen at bind time from among the following options:

- all bits 0,
- all bits 1,
- invalid value if possible (corresponding to Normalize_Scalars behaviour),
- a specified bit pattern.

Running the application or the application tests using different settings can detect more bugs. Any difference of behaviour between runs using different initial values are indications of the use of uninitialized values. Of course, this technique does not guarantee to find all bugs, but increases the chances of discovering them if the test coverage is wide enough. Furthermore, it is of particular value in the case where no invalid values exist, since the variation in behavior can indicate uninitialized values, even if all values are valid.

3.3 Additional Validity Checking

In GNAT version 3.13, only the RM mandated validity checkings were supported. They could be turned off, but there was no facility for forcing additional checks. In version 3.14, the notion of optional validity checks was added, and a switch with several levels was introduced as follows:

- gnatV0 → no checking,
- gnatV1 → RM checking,
- gnatV2 → check all assignment right hand sides.

The provision for checking assignment right hand sides was done precisely to improve validity checking in connection with the use of Normalize_Scalars. However, this turned out to miss many cases so an initial enhancement was provided that added two additional levels:

- gnatV3 → check all tests,
- gnatV4 → check all expressions.

Experimentation against Eurocontrol's applications showed that intermediate levels were needed. In particular, it was found that checking all ”in out” parameters caused difficulties, due to coding that was indeed incorrect with regard to validity checking, but that was in practice harmless in normal operation. Therefore, control over validity checking was eventually improved to allow specification of exactly which situations result in additional checks. Furthermore, a pragma Validity_Checks was introduced to allow control to be specified at the source level, and to be varied within a single unit. The possible settings in the final implementation of -gnatV (cf. GNAT User's Guide [7]) are as follows:

- gnatVa/n → turn on/off all validity checks (including RM),
- gnatVc/C → turn on/off checks for copies,
- gnatVd/D → turn on/off RM checks (on by default),
- gnatVf/F → turn on/off checks for floating-point,
- gnatVi/I → turn on/off checks for “in” params,
- gnatVm/M → turn on/off checks for “in out” params,
- gnatVo/O → turn on/off checks for operators,
- gnatVr/R → turn on/off checks for returns,
- gnatVs/S → turn on/off checks for subscripts,
- gnatVt/T → turn on/off checks for tests.

With the relevant checking mode on, the usage of an invalid value is detected and reported by raising a Constraint_Error exception.

3.4 Implementation in GNAT of Initialize_Scalars and gnatV

There were three considerations to the implementation of these new facilities in GNAT.

First, in order to ensure that Initialize_Scalars can be used on a selective basis, all possible cases of a client being compiled with or without the pragma, and the packages it uses being compiled with or without the pragma, must work. This involved the generation of some additional dummy initialization routines, which in practice are nearly always inlined, resulting in no additional run-time overhead.

Second, to allow the specification of invalid values at bind time, the code generated for Initialize_Scalars differs from that generated for Normalize_Scalars in that the values used to initialize otherwise uninitialized data are always copied from fixed memory locations, instead of being supplied as compile-time known constants as was done for Normalize_Scalars. This is slightly less efficient, but allows the memory locations to be modified at bind time. Indeed it would be possible to modify them at run time (e.g. by the use of an environment variable), and that is a planned future enhancement.

Finally, the insertion of additional validity checks required some care, because the compiler and its various optimization algorithms are quick to eliminate the additional checks on the grounds that they are obviously not required if the data is valid. Of course the whole point is that such checks are required because the data may be invalid, and the optimization procedures had to be modified to avoid the removal of needed tests. The actual validity checking was easily implemented, since the compiler already had mechanisms for the 'Valid attribute.

4 Application to Eurocontrol and obtained results

4.1 Bugs Discovered

GNAT has only reported real errors (uninitialized scalar usage). In other words, the Initialize_Scalars and gnatV checks have caused constraint errors only for uninitialized variables (no false positives). Not all the reported bugs had functional impacts, however, as the following example illustrates. GNAT reported a bug in code similar to:

```
Found : boolean;
Data  : A_Record_With_A_Status;

Find_A_Record (For_Key, Found, Data);

if Found and Data.Status = Not_Interesting then
  Found := False;
end if;
....
```

The procedure Find_A_Record sets Found to true if it finds a record corresponding to For_Key. If a record is found, Data is initialized otherwise it is not initialized. Some of the found records are however not interesting and must be filtered out. The above code is technically incorrect (an uninitialized Status is accessed in case Found is false) but this has no functional impact. One possible easy correction is to replace the “and” by an “and then”. Other corrections are of course possible (i.e. change the code in order to obtain a more elegant structure based on a cleaner specification of Find_A_Record).

When Initialize_Scalars was started to be used, the large majority of bugs were detected by the additional run-time checks on enumerated and boolean values. Most of these bugs are usually quite straightforward to correct once they are detected. Without Initialize_Scalars, they are however sometimes very tricky to detect. As an example, a bug was detected in a procedure that was waiting for an X protocol event to be received (up to a certain deadline). When the deadline was reached before asking X if an event occurred, the variable telling if an X event was pending was left uninitialized. This was then potentially leading to a call to X to handle the event even when the deadline was expired.

Float validation checking (gnatVf) has also detected quite a number of bugs in some numeric algorithms with unusual data (e.g. flight plans that do not respect the aircraft maximum performances). A lot of these bugs have been discovered by injecting massive amount of data. When floats are left uninitialized, conventional testing does not always reveal such bugs, as the massive results have to be analyzed in detail with respect to numerical correctness.

Finally, Initialize_Scalars and validation checking are not only discovering functional bugs. They also helped to detect efficiency bugs. E.g., a bug was discovered in a procedure that has to search in a list of (key, value) the value for a certain key. This was implemented on a generic list package providing a passive iterator. The boolean variable used to exit the iterator was left uninitialized. This had no functional effect as the return value was initialized properly even when the key was not found. When the key was found, however, potentially all the rest of the list was still traversed for no reason.

4.2 Incorrect Programming Idiom with “in out” Parameters

The following wrong programming idiom has lead to the need of fine grain control over validity checks resulting in the gnatV switch described previously.

```
procedure Read_Or_Write
   (Read_Mode : Boolean; A_Scalar : in out Natural) is
begin
   if Read_Mode then
      A_Scalar := ....; -- read from somewhere
   else
      Write (A_Scalar); -- write somewhere
   end if;
end Read_Or_Write;
```

```
N : Natural;
....
Read_Or_Write (Read_Mode => True, A_Scalar => N);
N := N + 2;
Read_Or_Write (Read_Mode => False, A_Scalar => N);
```

The idea is to write ‘low level’ symmetrical read/write procedures that can either read or write elementary types. Read/Write functions for composite types can then be programmed without needing to check if the operation to execute is the read or the write operation as this is delegated to the low level elementary type procedures. This is in fact a ’manual’ implementation of the `'Read/'Write` attributes of Ada95, designed and programmed at a time Ada83 was used at CFMU.

This idea looks attractive at first sight as it avoids programming and maintaining read and write procedure independently. When the compiler inserts a validity check for scalars parameters, then the above code is raising a constraint error whenever `Read_Mode` is True and `A_Scalar` is not initialized. We first started to correct this design error by splitting the code in separate read and write procedures for scalars. However, a lot of generics were expecting this kind of symmetrical procedures as generic parameter. As the same generics were instantiated with scalars or composite types, this kind of correction had a snowball effect, obliging to rework all the `Read_Or_Write` procedures.

A discussion was held with ACT, which resulted in a much finer control over what validity checks to insert. The case above was solved by having switches to specifically enable/disable the validation for in and in/out parameters. We are now disabling the validation for in/out parameters. This does not mean however that a real bug would not be discovered, only that it would be discovered not at the point of the wrong call but rather at the point where the called procedure would wrongly use the uninitialized scalar. In other words, when -gnatVM is used, a call to `Read_Or_Write` with `N` uninitialized does not raise constraint error, but a constraint error is raised inside `Read_Or_Write` when the `Write` procedure is called.

4.3 Performance Impact

The performance impact of tools influences how and when they can be used. The factors to be looked at are build time (compile, bind and link), the executable size and the run-time performance. Table 1 summarizes the impact of various combinations of GNAT switches on a representative CFMU test program. The ’mode’ column identifies switch combinations with a set of values amongst:

- 0 → no optimization,
- 2 → optimization (gcc -O2 + back-end inlining),
- i → Initialize_Scalars pragma,
- v → gnatVaM (all validation checks, except for in out parameters),
- r → reference manual checking (i.e. gnatVd) + integer overflow check,
- n → all validity checks off (including reference manual checkings).

Table 1. Performance impact of various switch combinations.

mode	current use	build time	executable size	run time
0r		100	100	100
0iv	development	118	107	160
2n		190	68	69
2r	operational	197	69	70
2iv		252	72	91

The numbers are relative to the 0r mode, which was the default development setup before the introduction of Initialize_Scalars. The impact of Initialize_Scalars and gnatVaM on the three factors is reasonable enough for daily usage by developers. The run-time penalty is however considered too costly for operational usage due to the high performance requirements of the CFMU application. Currently, the choice is to use the reference manual checks, which avoids the most horrible consequences of uninitialized scalars (erroneous execution) for a very small run-time penalty.

4.4 Running with Different Initial Values

CFMU intends to run the application test suite with different values to initialize the scalars. Typically, we will run the automatic regression tests with scalars bit patterns initialized to all 0 and all 1 (in addition to the currently used 'invalid values' initialization). This should help detect the uninitialized variables for which no invalid values can be generated (when the full bit pattern of the type is needed to represent all values).

Waiting for a future enhancement of the compiler, changing the default initial value implies to rebind the executable. As a temporary solution, CFMU has developed code to override the initial values at startup time, using a shell environment variable.

5 Conclusions

5.1 Usage During Development

With Initialize_Scalars and gnatV, a significant number of latent (and in some cases potentially serious) bugs were discovered in the large CFMU application. For all these cases, once the bug was discovered with the help of Initialize_Scalars, it was straightforward to pin-point the error in the code logic and ensure a correct initialization by modifying the logic or providing a required initial value. Detection of uninitialized variable usage is now done earlier in the development in an efficient and pragmatic manner. Based on this experience, we recommended the use of Initialize_Scalars and gnatVa as a default development mode when writing new code or enhancing existing code.

The GNAT compiler provides fine grain control over validity code insertion. This makes it possible to use Initialize_Scalars and validity checking on large existing applications that sometimes cannot be fully adapted to all the validity checkings provided.

5.2 Operational Test Usage

The Initialize_Scalars and additional checking level of GNAT has a limited impact on the code size, code performance and compile time. This impact is reasonable enough so that CFMU has transitioned to using Initialize_Scalars and gnatV checking as the default development mode. CFMU also intends to install for the first few weeks of operational evaluation the version compiled with Initialize_Scalars and gnatVaM. This is possible because the performance degradation in both memory and CPU usage is acceptable for limited capacity evaluation purposes.

Due to the very high performance requirements of the CFMU application, after a few weeks of operational test of this "checking version", the application compiled with optimization (i.e. with Ada RM checks and GNAT optimization switches on, but without Initialize_Scalars) will be installed. This optimized version, after a few more months of operational evaluation, will go operational.

5.3 Impact of Usage of Initialize_Scalars on How To Program

There is a trend in programming guidelines to "force" initializing everything at declaration resulting in code like:

```
B : Natural := 0;

if .... then
  B := 5;
else
  B := 8;
end if;
```

The difficulty with such an approach is that the initial value is meaningless. If this value is used accidentally, the results are potentially just as wrong as the use of an uninitialized value, and furthermore, the explicit initialization precludes the approach we have described in this paper, and thus may introduce bugs that are much harder to find and fix. The automatic initialization under control of the compiler using Initialize_Scalars is a far preferable approach.

We therefore recommend that when a scalar is declared, the programmer should avoid initializing it if the code is supposed to set the value on all paths. It is better to let Initialize_Scalars + gnatVa detect the bug in the code logic rather than trying to deal with meaningless initial values. Even for safety-critical programs, we can first compile with Initialize_Scalars + gnatVa + invalid values and then, if needed, field the code with Initialize_Scalars + all zero values (if it is the case that zero values give the code a better chance of avoiding seriously improper behavior).

References

[1] Waroquiers, P.; *Ada Tasking and Dynamic Memory: To Use or Not To Use, That's a Question!*, Proceedings of International Conference on Reliable Software Technologies - Ada Europe 1996, Montreux, Switzerland, June 10–14, 1996, Alfred Strohmeier (Ed.), Lecture Notes in Computer Science, vol. 1088, Springer-Verlag, 1996, pp.460–470.

[2] Barnes, J.; *High Integrity Ada; The Spark Approach*, Addison Wesley, 1997.

[3] Waroquiers, P., Van Vlierberghe, S., Craeynest, D., Hately, A., and Duvinage, E.; *Migrating Large Applications from Ada83 to Ada95*, Proceedings of International Conference on Reliable Software Technologies - Ada Europe 2001, Leuven, Belgium, May 14–18, 2001, Dirk Craeynest, Alfred Strohmeier (Eds.), Lecture Notes in Computer Science, vol. 2043, Springer-Verlag, 2001, pp.380–391.

[4] *Purify on-line Unix manual*, Rational Software Corporation, June 2000.

[5] Taft, S.T., Duff, R.A., Brukardt, R.L. and Plödereder, E.; *Consolidated Ada Reference Manual. Language and Standard Libraries*, ISO/IEC 8652:1995(E) with COR.1:2000, Lecture Notes in Computer Science, vol. 2219, Springer-Verlag, 2001.

[6] *GNAT Reference Manual - The GNU Ada95 Compiler*, Version 3.15a, Ada Core Technologies, 30 January 2002.

[7] *GNAT User's Guide for Unix Platforms*, Version 3.15a, Ada Core Technologies, 30 January 2002.