AdaCore
The GNAT Pro Company

Source Code to Object Code Traceability Study

Release 1.0

Document ID: TEC.PB04-045, Issue 1.0

December 16, 2016

CONTENTS

CHAPTER

TWO

INTRODUCTION

2.1 Context

Approval of aviation software to the guidance of DO-178B/ED-12B and DO-178C/ED-12C [DO178C] requires an applicant to assess the correspondence between source code and object code in certain circumstances. In particular, for Level A software, source code to object code traceability must be established (see paragraph 6.4.4.2b of DO-178C/ED-12C). When the compiler generates object code that is not directly traceable to the source code, additional verifications must be performed. As proposed in the position paper CAST-12 [CAST12], an acceptable approach is to identify that untraceable compiler-generated object code and verify it.

2.1.1 Compiler Identification

This document analyzes the code generated by the GNAT Pro High-Integrity Edition 7.4.3 compiler hosted on Windows and targeting PowerPC e500v2 ELF identified by the string (issued when using the `--version` switch):

```
powerpc-eabispe-gcc (GCC) 4.9.4 20151030 (for GNAT Pro 7.4.3 20161026)
```

2.1.2 Compilation Options and Restrictions

The analysis performed in this document is based on the compilation options and restrictions defined in this section. Therefore, the actual selected options and coding standard used by the applicant must be compliant with those provided in this section.

The set of compilation options used in this study has been selected in order to reduce source code to object code traceability issues, as well as reflect the needs of the customer. They are:

- `-O1`. This optimization level reduces code size and execution time (by eliminating redundant code).
- `-gnatn`. This flag, together with `-O1`, allows inlining of subprograms marked explicitly (with `pragma Inline` or `pragma Inline_Always`) if they are suitable for inlining (small enough and not containing something that the compiler cannot support in inlined subprograms).
- `-ffunction-sections`. Place each function into its own section in the object code, for a potential linker optimization to improve locality of reference in the instruction space.
- `-fdata-sections`. Place each data item into its own section in the object code, for a potential linker optimization to improve locality of reference in the instruction space.
- `-gnatp`. Suppress all language-defined checks.
- `-mstrict-align`. Unaligned memory references are not handled by the system.
- `-gnat12`. Allow full Ada 2012 features.
- `-mno-sdata`. Put all initialized global and static data in the '.data' section, and all uninitialized data in the '.bss' section.

- `-mno-spe`. Disable the generation of SPE simd instructions.
- `-mfloat-gprs=single`. Enable the use of single-precision floating-point operations, and disable the use of double-precision floating-point operations.

The Coding Standard also uses some other options which do not have any influence on code generation, used for generating extra information for debugging (`-g`), for compiling only (`-c`), for controlling the behavior in case of compile-time errors (`-gnatf`, `-gnatQ`), for enabling some warning modes (`-gnatwa`, `-gnatwt`, `-gnatw.X`, `-gnatwl`), for assembly listings (`-mregnames`, `-fverbose-asm`), and for enabling some style checks (`-gnatye`, `-gnatyk`, `-gnaty0`). Their presence or absence is hence irrelevant for the purpose of analyzing code generation.

The run-time to use is the Zero FootPrint run-time. There is a set of restrictions, excluded predefined packages, and configuration pragmas, which are imposed by the use of this run time, and are defined in [GNATHIE]. The study is valid for run-times that are compatible with the zfp-p2020 runtime provided by GNAT. A run-time is compatible if it enforces the same set of restrictions and if it uses the same System package.

2.1.3 Link Control

The GNAT Pro toolchain provides the mechanisms to control the libraries that are linked into the final executable. The `-nostdlib` switch prevents linking with standard libraries (such as the C library `libgcc.a`) that are otherwise linked in implicitly. Therefore, the use of the `-nostdlib` switch is recommended, as well as explicitly adding a reference to the concrete library that is to be linked in, if needed, with a `<path>/libgcc.a` linker option. This mechanism allows the user to control the libraries that are linked into the final executable and therefore those for which certification material would be required.

The `-nostdlib` option needs to be passed to both the binder and the linker in order to avoid using standard libraries. Passing the option to the builder (`gnatmake`, `gprbuild`) will pass this option to both the binder and to the linker.

2.2 Strategy

The intent of structural coverage analysis, as defined in section 6.4.4.2 of DO-178B/ED-12B and DO-178C/ED-12C [DO178C], is to determine which code structures were not exercised by the requirements-based tests. When the software level is A, and the compiler, linker, or other means generate branches or side effects that are not immediately apparent at the source code level, then additional verification must be performed on the object code to establish the correctness of untraceable object code.

The position paper CAST-12 [CAST12] is used as guideline for this source code to object code traceability study.

2.2.1 Source to Object Code Traceability Analysis

The first step of the analysis is to perform the source to object code traceability analysis. Object code is considered as directly traceable if its structure is equivalent to the corresponding source code, meaning that the structural coverage analysis performed on the source code is sufficient to assess the coverage of the object code by requirements-based testing.

Object code is therefore considered not directly traceable when compiler-added code implements certain features that change the structure of the generated object with respect to the source code. Particular attention is paid to unexpected jump instructions and calls to external routines. In this case, the structural coverage analysis of the source code does not ensure that the added code has been fully exercised, and therefore additional verification activities need to be performed in order to assess its correctness.

When compiler-added code generates calls to operating system routines (or to routines to be provided by the user in order to support extended functionality such as dynamic memory), they are identified, recorded, and listed in the report, see chapter *External Calls Index*.

Given the importance of jump and call instructions when analyzing the structure of the object code, these instructions are highlighted using bold fonts in all assembly code listings.

The traceability analysis is performed on representative source code and object code. The coding standard, compiler version and options, and restrictions for the analysis are specified in section *Context*.

Representative source code is obtained by traversing the Ada Language Reference Manual [Ada12] and writing code samples for every particular feature that is allowed by the coding standard, the run-time library, and the restrictions being used. This analysis can be found in section *Ada Features Survey*.

Beyond this verification of Ada constructs taken in isolation, section *Combined Ada Features Survey* provides an analysis of code samples which combine elementary Ada features in various ways.

These code samples are compiled, and the generated assembly code is examined to verify its mapping to the original source code. The results of the traceability inspection is supplied, and additional comments are provided when untraceable code or hidden calls are detected.

2.2.2 Additional Verifications for Untraceable Code

When untraceable code is found, additional verification is performed to establish the correctness of the generated code sequences (see *Additional Verifications*). This activity relies on the following steps:

1. Requirement definition for compiler-added code

   The intended behavior of compiler-added code is defined. This definition is expressed in terms of requirements.

2. Definition of test cases

   Test cases are developed in order to verify the compliance of this added code against the corresponding requirements.

3. Definition of test procedures

   Test procedures are developed from the test cases, including the definition of the set of inputs and the expected results.

4. Execution

   The construct generating the untraceable code is compiled along with its tests in order to produce an executable. It is executed and the results are collected.

   The executable is composed of two different parts: one is the code representative of the embedded software that we want to exercise and cover, containing the construct that we are verifying, and compiled with the options and restrictions that are specified in *Compilation Options and Restrictions*; the other part is code that acts as a test harness, which is there to create an executable that can verify the requirements and coverage objectives, and which is therefore written without the restrictions imposed on the embedded application.

5. Analysis of the results

   The results of the execution are compared against the expected ones defined in the test cases. Each execution provides a report that indicates whether the test passed or failed.

6. Completeness

   Thorougness of the requirement-based tests is completed by analyzing the generated code: this analysis checks that added code is understood and can be traced back to test cases. This manual analysis is done on generated code annotated by instruction coverage; note however that instruction coverage does not replace this activity.

   Instruction coverage (at object level) is performed using the GNATcoverage tool chain. The GNATemulator instrumented emulation platform first generates execution traces. GNATcoverage then extracts coverage information from these traces.

2.3 Structure of the Document

Chapters *Ada Features Survey* and *Combined Ada Features Survey* contain the analysis of the object code generated for representative Ada 2012 constructs:

- Chapter *Ada Features Survey* identifies code samples exercising the Ada features defined in the different subsections of the Ada 2012 Reference Manual [Ada12]. For traceability purposes, the section numbers in this chapter correspond to those in the Reference Manual. When the subsection is only informative, or when the coding standard or the run-time used does not allow the use of the defined construct, there is a justification explaining the reason why there is no code sample.
- Chapter *Combined Ada Features Survey* supplements the previous one by identifying code samples which are the result of combining elementary Ada features.

For each of these subsections, the following information is provided:

- Description. Outline of the Ada feature being tested.
- Traceability analysis result. Source code to object code traceability analysis, indicating "directly traceable code" or "untraceable code". This may be followed by a description or explanation, particularly when there is untraceable code to be analyzed.
- Ada code. It contains an excerpt of the compilable unit that is used to analyze the Ada feature. The full Ada sources are available in a separate package.
- ASM-generated code. This part shows the assembly code that is generated for the Ada unit. Jump and call instructions use bold fonts to highlight the structure of the object code.
- References to additional verifications. When object code that is not directly traceable to the source code is found, this section contains references to the additional verifications that are done in chapter *Additional Verifications*.
- External functions. When hidden calls to external functions are present, this section summarizes them and points to their entry in chapter *External Calls Index*.

The last two sections mentioned above are only present when the test case being analyzed has untraceable code or calls to external functions respectively.

Chapter *Additional Verifications* includes the analysis being done on object code that is not directly traceable to source code. The structure of the information provided is:

- Description. Outline of the Ada feature being tested.
- Requirements and test cases. This section describes the intended behavior of the compiler-added code being analyzed, with a set of test cases designed to ensure proper requirements-based testing.
- Test source code. This contains the excerpt of the compilable units that is relevant to understand the test case and test procedure. The full Ada sources are available in a separate package.
- Expected results. This is the expected output of the set of test cases devised for requirements-based testing. These results come in the form of assertion messages inserted in the Ada code to assess whether the requirements are met.
- Results of requirements-based testing. This section contains the actual result of the execution of the test cases.
- Object-level coverage analysis. Result of the instruction coverage analysis. A "+" mark means that the instruction has been executed, and a "-" identifies any instruction that has **not** been exercised.
- Additional verification result. Analysis of requirements-based testing and structural coverage results.

Chapter *Conclusion* summarizes the activities and results of this study, and the conditions under which the applicant may perform the structural coverage analysis on source code.

Finally, annex *External Calls Index* identifies and lists the set of routines that may be implicitly called by compiler-added code.

CHAPTER

THREE

ADA FEATURES SURVEY

This chapter contains the analysis of representative source code obtained by traversing the Ada 2012 Reference Manual [Ada12]. Code samples are written for every particular feature that is allowed by the coding standard, the run-time library, and the restrictions being used.

These code samples are compiled, and the generated assembly code is examined to verify its mapping to the original source code. The results of the traceability inspection are recorded, and additional comments are provided when untraceable code or hidden calls are detected.

3.1 General

This section is an introduction to the Ada programming language, so it contains no Ada feature to test.

3.2 Lexical Elements

This section describes the lexical elements that make up the Ada language, so it contains no Ada feature to test.

3.3 Declarations

3.3.1 Declarations

This subsection defines the entities that are declared by declarations, so it contains no Ada feature to test.

3.3.2 Types and Subtypes

This subsection describes basic characteristics of types and subtypes that are verified in subsequent subsections of this chapter for each individual kind of type. Hence, no specific test has been defined for this subsection, with the exception of subtype predicates.

3.3.2.1 Subtype Predicates

Traceability Case ch030204_01

Description

Subtype predicates - conversion in parameter passing (3.2.4 (31/3)).

Ada Code

```
procedure T1_Out_Mode (Obj1 : out T1) is
begin
   Obj1 := An_Integer;
end T1_Out_Mode;

procedure T2_Out_Mode (Obj2 : out T2) is
begin
   Obj2 := An_Integer;
end T2_Out_Mode;

function Get_T1 return T1 is
begin
   return An_Integer;
end Get_T1;

function Get_T2 return T2 is
begin
   return An_Integer;
end Get_T2;

procedure Test_T1_In_Mode (Obj1 : Integer) is
begin
   ch030204_01types.T1_In_Mode (Obj1);
end Test_T1_In_Mode;

procedure Test_T2_In_Mode (Obj2 : Integer) is
begin
   ch030204_01types.T2_In_Mode (Obj2);
end Test_T2_In_Mode;

procedure Test_T1_In_Out_Mode (Obj1 : in out Integer) is
begin
   ch030204_01types.T1_In_Out_Mode (Obj1);
end Test_T1_In_Out_Mode;

procedure Test_T2_In_Out_Mode (Obj2 : in out Integer) is
begin
   ch030204_01types.T2_In_Out_Mode (Obj2);
end Test_T2_In_Out_Mode;

subtype T1 is Integer with
  Static_Predicate  => T1 in 1 | 13;

subtype T2 is Integer with
  Dynamic_Predicate => Check_T (T2);
```

Assembly Code

```
ch030204_01__t1_out_mode:
  lis %r9,.LANCHOR0@ha      # tmp159,
  lwz %r3,.LANCHOR0@l(%r9)      # ch030204_01__an_integer,
  blr
ch030204_01__t2_out_mode:
  lis %r9,.LANCHOR0@ha      # tmp159,
  lwz %r3,.LANCHOR0@l(%r9)      # ch030204_01__an_integer,
  blr
ch030204_01__get_t1:
  lis %r9,.LANCHOR0@ha      # tmp159,
  lwz %r3,.LANCHOR0@l(%r9)      # ch030204_01__an_integer,
  blr
ch030204_01__get_t2:
  lis %r9,.LANCHOR0@ha      # tmp159,
  lwz %r3,.LANCHOR0@l(%r9)      # ch030204_01__an_integer,
```

```
  blr
ch030204_01__test_t1_in_mode:
  stwu %r1,-8(%r1)       #,,
  mflr %r0       #,
  stw %r0,12(%r1)       #,
  bl ch030204_01types__t1_in_mode       #
  lwz %r0,12(%r1)       #,
  mtlr %r0       #,
  addi %r1,%r1,8       #,,
  blr       #
ch030204_01__test_t2_in_mode:
  stwu %r1,-8(%r1)       #,,
  mflr %r0       #,
  stw %r0,12(%r1)       #,
  bl ch030204_01types__t2_in_mode       #
  lwz %r0,12(%r1)       #,
  mtlr %r0       #,
  addi %r1,%r1,8       #,,
  blr       #
ch030204_01__test_t1_in_out_mode:
  stwu %r1,-8(%r1)       #,,
  mflr %r0       #,
  stw %r0,12(%r1)       #,
  bl ch030204_01types__t1_in_out_mode       #
  lwz %r0,12(%r1)       #,
  mtlr %r0       #,
  addi %r1,%r1,8       #,,
  blr       #
ch030204_01__test_t2_in_out_mode:
  stwu %r1,-8(%r1)       #,,
  mflr %r0       #,
  stw %r0,12(%r1)       #,
  bl ch030204_01types__t2_in_out_mode       #
  lwz %r0,12(%r1)       #,
  mtlr %r0       #,
  addi %r1,%r1,8       #,,
  blr       #
```

Traceability Analysis Result

Directly traceable code. Subprograms are generated for subtype predicates, for cases where they are evaluated: ‘Valid attribute and membership tests. These cases are considered in separate tests. For the other subprograms, and as checks are disabled, no branch is generated.

3.3.5.4 Integer Types

Traceability Case ch030504_01

Description

Attributes for a signed integer type (static case).

Ada Code

```
type Integer32 is range -(2 **31) .. +(2 **31 - 1);
for Integer32'Size use 32;

First_I32  : Integer32 := Integer32'First;
Last_I32   : Integer32 := Integer32'Last;

First_Base : constant  := Integer32'Base'First;

Min_I32    : Integer32 := Integer32'Min (23, 32);
Max_I32    : Integer32 := Integer32'Max (23, 32);

Succ_I32   : Integer32 := Integer32'Succ (23);
Pred_I32   : Integer32 := Integer32'Pred (23);
```

Assembly Code

No code was generated in the .text section for this test.

Traceability Analysis Result

No object code generated for this feature, as expected.

Traceability Case ch030504_06

Description

Use of unsigned integer operations from package Interfaces.

Ada Code

```
Sl8  : U8 := Shift_Left (X8, Amount);
Sr8  : U8 := Shift_Right (X8, Amount);
Sra8 : U8 := Shift_Right_Arithmetic (X8, Amount);
Rl8  : U8 := Rotate_Left (X8, Amount);
Rr8  : U8 := Rotate_Right (X8, Amount);
```

Assembly Code

```
ch030504_06___elabs:
  lis %r9,temps__tint@ha      # tmp170,
  lwz %r9,temps__tint@l(%r9)      # temps__tint, D.2350
  lis %r10,.LANCHOR0@ha      # tmp172,
  stw %r9,.LANCHOR0@l(%r10)      # ch030504_06__amount, D.2350
  rlwinm %r8,%r9,0,0xff      # D.2349, D.2350
  lis %r10,.LANCHOR1@ha      # tmp174,
  stb %r8,.LANCHOR1@l(%r10)      # ch030504_06__x8, D.2349
  cmpwi %cr7,%r9,7      #, tmp175, D.2350
  bgt- %cr7,.L2      #
  slw %r10,%r8,%r9      # tmp178, D.2349, D.2350
  lis %r7,.LANCHOR2@ha      # tmp177,
  stb %r10,.LANCHOR2@l(%r7)      # ch030504_06__sl8, tmp178
  srw %r10,%r8,%r9      # tmp179, D.2349, D.2350
  rlwinm %r10,%r10,0,0xff      # D.2349, tmp179
.L3:
  lis %r7,.LANCHOR3@ha      # tmp181,
  stb %r10,.LANCHOR3@l(%r7)      # ch030504_06__sr8, D.2349
  cmpwi %cr7,%r9,7      #, tmp186, D.2350
  li %r7,7      #,
  isel %r10,%r7,%r9,29      # D.2353,, D.2350,
  extsb %r7,%r9      # D.2350, D.2350
  extsb %r10,%r10      # D.2353, D.2353
  sraw %r10,%r7,%r10      # tmp189, D.2350, D.2353
  lis %r7,.LANCHOR4@ha      # tmp183,
  stb %r10,.LANCHOR4@l(%r7)      # ch030504_06__sra8, tmp189
  srawi %r10,%r9,3      # tmp190, D.2350,
  addze %r10,%r10      # tmp190
  slwi %r10,%r10,3      # tmp191, tmp190,
  subf %r9,%r10,%r9      # D.2353, tmp191, D.2350
  slw %r7,%r8,%r9      # tmp195, D.2349, D.2353
  neg %r10,%r9      # tmp199, D.2353
  rlwinm %r10,%r10,0,29,31      # tmp198, tmp199,
  srw %r10,%r8,%r10      # tmp197, D.2349, tmp198
  or %r10,%r7,%r10      #, tmp202, tmp195, tmp197
  lis %r7,.LANCHOR5@ha      # tmp193,
  stb %r10,.LANCHOR5@l(%r7)      # ch030504_06__rl8, tmp202
  srawi %r10,%r9,3      # tmp207, D.2353,
  addze %r10,%r10      # tmp207
  slwi %r10,%r10,3      # tmp208, tmp207,
  subf %r9,%r10,%r9      # D.2353, tmp208, D.2353
  srw %r10,%r8,%r9      # tmp211, D.2349, D.2353
  neg %r9,%r9      # tmp214, D.2353
  rlwinm %r9,%r9,0,29,31      # tmp213, tmp214,
  slw %r9,%r8,%r9      # tmp212, D.2349, tmp213
  or %r9,%r10,%r9      #, tmp217, tmp211, tmp212
  lis %r10,.LANCHOR6@ha      # tmp204,
  stb %r9,.LANCHOR6@l(%r10)      # ch030504_06__rr8, tmp217
  blr
```

```
.L2:
  li %r7,0       # tmp220,
  lis %r10,.LANCHOR2@ha       # tmp219,
  stb %r7,.LANCHOR2@l(%r10)       # ch030504_06__s18, tmp220
  li %r10,0       # D.2349,
  b .L3       #
```

Traceability Analysis Result

Untraceable code. Use of the Shift_Left and Shift_Right operations results in the generation of conditional code to handle the special case where the shift amount value is greater than or equal to the size of the type. At least one untraceable conditional branch is present in the object code.

Additional verifications

- *SHIFT_LEFT*
- *SHIFT_RIGHT*

3.4.5.5 Multiplying Operators

Traceability Case ch040505_01

Description

Multiplication and division operators on integers.

Ada Code

```
Res1 := Int1 * Int2;
Res2 := Int1 / Int2;
```

Assembly Code

```
_ada_ch040505_01:
  mr %r9,%r3       # int1, int1
  mullw %r10,%r3,%r4       # tmp161, int1, int2
  li %r3,0       # D.2322,
  rlwimi %r3,%r10,24,0,7       # D.2322, tmp161,,,
  divw %r9,%r9,%r4       # tmp163, int1, int2
  rlwimi %r3,%r9,16,8,15       # D.2322, tmp163,,,
  blr
```

Traceability Analysis Result

Directly traceable code. No branch is generated.

Traceability Case ch040505_02

Description

Mod operator for signed integers.

Ada Code

```
return A mod B;
```

Assembly Code

```
_ada_ch040505_02:
  cmpwi %cr7,%r4,-1       #, tmp160, b
  beq- %cr7,.L8       #
  cmpwi %cr7,%r4,0       #, tmp164, b
  blt- %cr7,.L4       #
  cmpwi %cr7,%r3,0       #, tmp165, a
  blt- %cr7,.L3       #
  divw %r9,%r3,%r4       # tmp162, a, b
  b .L7       #
.L3:
  addi %r9,%r3,1       # a, a,
  b .L6       #
.L4:
  addi %r9,%r3,-1       # a, a,
  cmpwi %cr7,%r3,0       #, tmp166, a
  bgt+ %cr7,.L6       #
  divw %r9,%r3,%r4       # tmp162, a, b
  b .L7       #
.L6:
  divw %r9,%r9,%r4       # tmp162, a, b
  addi %r9,%r9,-1       # tmp162, tmp162,
.L7:
  mullw %r4,%r9,%r4       # tmp167, tmp162, b
  subf %r3,%r4,%r3       # D.2323, tmp167, a
  blr
.L8:
  li %r3,0       # D.2323,
  blr
```

Traceability Analysis Result

Untraceable code. Extra code is generated to handle the case of B=-1 in order to guard against a possible overflow when A=Integer'First, and to normalize the result when A and B have different signs. Untraceable conditional branches are present in the object code.

Additional verifications

- *MOD*

Traceability Case ch040505_03

Description

Signed Integer rem.

Ada Code

```
return A rem B;
```

Assembly Code

```
_ada_ch040505_03:
  cmpwi %cr7,%r4,-1       #, tmp160, b
  beq- %cr7,.L3       #
  divw %r9,%r3,%r4       # tmp163, a, b
  mullw %r4,%r9,%r4       # tmp164, tmp163, b
  subf %r3,%r4,%r3       # D.2323, tmp164, a
  blr
.L3:
  li %r3,0       # D.2323,
  blr
```

Traceability Analysis Result

Untraceable code. Extra code is generated to handle the case of B=-1 in order to guard against a possible overflow when A=Integer'First. Untraceable conditional branches are present in the object code.

Additional verifications

- *REM*

Traceability Case ch040505_04

Description

Rem operator for positive integers.

Ada Code

```
R1 :=  N1 rem N2;
R2 :=  N3 rem N4;
R3 :=  N5 rem N6;
```

Assembly Code

```
_ada_ch040505_04:
  divwu %r9,%r3,%r4       # tmp169, n1, n2
  mullw %r4,%r9,%r4       # tmp170, tmp169, n2
  subf %r9,%r4,%r3       # tmp171, tmp170, n1
  li %r3,0       # D.2328,
  rlwimi %r3,%r9,24,0,7       # D.2328, tmp171,,,
  divwu %r9,%r5,%r6       # tmp175, n3, n4
  mullw %r6,%r9,%r6       # tmp176, tmp175, n4
  subf %r5,%r6,%r5       # tmp177, tmp176, n3
  rlwimi %r3,%r5,0,16,31       # D.2328, tmp177,,,
  divwu %r9,%r7,%r8       # tmp181, n5, n6
  mullw %r8,%r9,%r8       # tmp182, tmp181, n6
  subf %r4,%r8,%r7       # tmp3, tmp182, n5
  blr
```

Traceability Analysis Result

Directly traceable code. The “rem” operator for types whose range is known at compile time to contain only positive or null values does not generate code to handle negative operators.

CHAPTER

FOUR

COMBINED ADA FEATURES SURVEY

This chapter contains the analysis of code samples which are the result of combining elementary Ada features.

These code samples are compiled, and the generated assembly code is examined to verify its mapping to the original source code. The results of the traceability inspection are recorded, and additional comments are provided when untraceable code or hidden calls are detected.

4.1 Mix different assignments

Traceability Case mixing_assignment

Description

This test case includes the composition of the following elementary test cases: assignment for modular types, enumerate types, and record types.

Ada Code

```
procedure Mixing_Assignment
   (Par_91_l : in out Types.Mod8;
    Par_91_r : Types.Mod8;
    Par_93_l : in out Types.Enum;
    Par_93_r : Types.Enum;
    Par_95_l : in out Types.Simple_Record;
    Par_95_r : Types.Simple_Record)
is
   use Types;
begin
   Par_91_l := Par_91_r;
   Par_93_l := Par_93_r;
   Par_95_l := Par_95_r;
end Mixing_Assignment;
```

Assembly Code

```
_ada_mixing_assignment:
  lwz %r10,0(%r8)       # *par_95_r_5(D), *par_95_r_5(D)
  lwz %r9,4(%r8)       # *par_95_r_5(D), *par_95_r_5(D)
  stw %r10,0(%r7)       # *par_95_l_4(D), *par_95_r_5(D)
  stw %r9,4(%r7)       # *par_95_l_4(D), *par_95_r_5(D)
  li %r3,0      # D.2331,
  rlwimi %r3,%r4,24,0,7       # D.2331, par_91_r,,,
  rlwimi %r3,%r6,16,8,15       # D.2331, par_93_r,,,
  blr
```

Traceability Analysis Result

The composition of these constructs generates code which is equivalent to the composition of the object code

generated for modular types (*Integer Types*), enumerate types (*Enumeration Types*), and record types (*Record Types*).

4.2 Nest block statements

Traceability Case block_nesting

Description

This test case includes the composition of the following elementary test cases: (nested) if_statement and (nested) blocks.

Ada Code

```
procedure Block_Nesting
   (Par_32 : Boolean;
    Par_28 : Integer;
    Par_30 : in out Integer)
is
begin
   BLOCK_Par_28 : declare
      I : Integer := Par_28;
   begin
      BLOCK_Par_30 : declare
         J : Integer := Par_30;
      begin
         Par_30 := I;
         if Par_32 then
            Par_30 := I + J;
         end if;
      end BLOCK_Par_30;
   end BLOCK_Par_28;
end Block_Nesting;
```

Assembly Code

```
_ada_block_nesting:
  add %r5,%r4,%r5       # tmp162, par_28, par_30
  cmpwi %cr7,%r3,0       #, tmp163, par_32
  isel %r3,%r4,%r5,30       #, par_28, tmp162,
  blr
```

Traceability Analysis Result

The composition of these constructs generates code which is equivalent to the composition of the object code generated for if_statement (*If Statements*) and blocks (*Block Statements*).

4.5 Nest for statements

Traceability Case for_nesting

Description

This test case includes the composition of the following elementary test cases: (nested) for_loops, assignment for modular types, and addition for modular types.

Ada Code

```
procedure For_Nesting
   (Par_16_l   : in out Types.Mod8;
    Par_16_r   : Types.Mod8;
    Par_14_Res : out Types.Mod8;
    Par_14_l   : Types.Mod8;
    Par_14_r   : Types.Mod8;
    Par_12     : Positive;
    Par_13     : Positive)
is
   use Types;
begin
   Par_14_Res := 0;
   for Iterator_Par_12 in 0 .. Par_12 loop
      for Iterator_Par_13 in 0 .. Par_13 loop
         Par_14_Res := Par_14_Res + Par_14_l + Par_14_r;
         Par_16_l := Par_16_r;
      end loop;
   end loop;
end For_Nesting;
```

Assembly Code

```
_ada_for_nesting:
  addi %r9,%r8,1       # tmp173, par_13,
  add %r6,%r5,%r6       # tmp174, par_14_l, par_14_r
  mullw %r6,%r9,%r6       # tmp177, tmp173, tmp174
  rlwinm %r5,%r6,0,0xff       # D.2353, tmp177
  li %r6,-1       # iterator_par_12,
  li %r10,0       # par_14_res,
  li %r3,-1       # iterator_par_13,
.L3:
  addi %r6,%r6,1       # iterator_par_12, iterator_par_12,
  mr %r9,%r3       # iterator_par_13, iterator_par_13
  subf %r11,%r3,%r8       #, iterator_par_13, D.2354
  mtctr %r11       # tmp184,
.L2:
  addi %r9,%r9,1       # iterator_par_13, iterator_par_13,
  bdnz .L2       #
  add %r10,%r10,%r5       # tmp179, par_14_res, D.2353
  rlwinm %r10,%r10,0,0xff       # par_14_res, tmp179
  cmpw %cr7,%r6,%r7       # D.2354, tmp180, iterator_par_12
  bne+ %cr7,.L3       #
  li %r3,0       # D.2338,
  rlwimi %r3,%r4,24,0,7       # D.2338, par_16_r,,,
  rlwimi %r3,%r10,16,8,15       # D.2338, par_14_res,,,
  blr
```

Traceability Analysis Result

The composition of these constructs generates code which is equivalent to the composition of the object code generated for for_loops (*Loop Statements*) and modular types (*Integer Types*).

4.9 Nest if within case statements

Traceability Case case_if_nesting

Description

This test case includes the composition of the following elementary test cases: (nested) if_statements, case_statements, and assignment for modular types.

Ada Code

```
procedure Case_If_Nesting
   (Par_66   : Boolean;
    Par_67_l : in out Types.Mod8;
    Par_67_r : Types.Mod8;
    Par_65   : Types.Enum)
 is
   use Types;
begin
   case Par_65 is
      when Types.One =>
         if Par_66 then
            Par_67_l := Par_67_r;
         end if;
      when others => null;
   end case;
end Case_If_Nesting;
```

Assembly Code

```
_ada_case_if_nesting:
  cmpwi %cr7,%r6,0      #, tmp160, par_65
  bne- %cr7,.L2      #
  cmpwi %cr7,%r3,0      #, tmp163, par_66
  isel %r4,%r4,%r5,30      # par_67_l, par_67_l, par_67_r,
.L2:
  mr %r3,%r4      #, par_67_l
  blr
```

Traceability Analysis Result

The composition of these constructs generates code which is equivalent to the composition of the object code generated for if_statements (*If Statements*), case_statements (*Case Statements*), and modular types (*Integer Types*).

CHAPTER

FIVE

ADDITIONAL VERIFICATIONS

This chapter contains the additional verification performed when untraceable code is found, in order to establish the correctness of the generated code sequences. The following activities are performed:

1. Requirement definition for compiler-added code

   The intended behavior of compiler-added code is defined. This definition is expressed in term of requirements.

2. Definition of test cases

   Test cases are developed in order to verify the compliance of this added code against the corresponding requirements.

3. Definition of test procedures

   Test procedures are developed from the test cases, including the definition of the set of inputs and the expected results.

4. Execution

   The construct generating the untraceable code is compiled along with its tests in order to produce an executable. It is executed and the results are collected.

   The executable is composed of two different parts: one is the code representative of the embedded software that we want to exercise and cover, containing the construct that we are verifying, and compiled with the options and restrictions that are specified in *Compilation Options and Restrictions*; the other part is code that acts as a test harness, which is there to create an executable that can verify the requirements and coverage objectives, and which is therefore written without the restrictions imposed on the embedded application, allowing us to exercise, for example, code generated for checks that would otherwise be forbidden.

5. Analysis of the results

   The results of the execution are compared against the expected ones defined in the test cases. Each execution provides a report that indicates whether the test passed or failed.

6. Instruction coverage analysis

   Once all the tests are passed, meaning that the behavior of the added code is compliant with its requirements, instruction coverage analysis is performed on the object code. The objective of this analysis is to assess the thoroughness of the requirements-based tests. When not all assembly instructions of the tested Ada constructs have been exercised, a justification is given.

   This instruction coverage analysis (at object level) is performed using the GNATcoverage tool chain. The GNATemulator instrumented emulation platform first generates execution traces. GNATcoverage then extracts coverage information from these traces.

5.15 Remainder operator

Test procedure test_rem

Description

Requirements for `x REM y`. Note that `REM` is not defined if y = 0.

Requirements and test cases

- Requirement REM.1: When x >= 0 & y > 0 then x rem y = x - (x/y)*y
  - Test case REM.1.1: Use positive values for both operands and the result will be a positive value that is equal to x - (x/y)*y.
- Requirement REM.2: When x <= 0 and y > 0 then x rem y = -(-x rem y)
  - Test case REM.2.1: Use a negative value for the left operand and a positive one for the right operand. The result will be a negative value whose absolute value is the same as when using the absolute value of the left operand.
- Requirement REM.3: When y < 0 then x rem y = x rem -y
  - Test case REM.3.1: Use a positive value for the left operand and a negative one for the right operand. The result will be a positive value equal to the result of then rem operation with both operands positives.
  - Test case REM.3.2: Use negative values for both operands and the result will be a negative value whose absolute value is the same as with both operands positive.
- Requirement REM.4: When y = -1, forall x then x rem y = 0 (avoids overflow when x = int'First)
  - Test case REM.4.1: Use a positive left operand and -1 as the right operand. The expected result of this operation is 0.
  - Test case REM.4.2: Verify that there is no overflow when the left operand is Integer'First and the right operand is -1. The expected result of this operation is 0.

Test source code

```
Val1 : Integer := 83;
Val2 : Integer := 13;
Res1 : Integer := Val1 - (Val1 / Val2) * Val2;

--  Subprogram under test
function My_Rem (A, B : Integer) return Integer is
begin
   return A rem B;
end My_Rem;

--  Test procedure

Assert (My_Rem (Val1,         Val2) =  Res1, "REM.1.1");
Assert (My_Rem (-Val1,        Val2) = -Res1, "REM.2.1");
Assert (My_Rem (Val1,        -Val2) =  Res1, "REM.3.1");
Assert (My_Rem (-Val1,       -Val2) = -Res1, "REM.3.2");
Assert (My_Rem (Val1,           -1) =     0, "REM.4.1");
Assert (My_Rem (Integer'First, -1) =     0, "REM.4.2");
```

Expected results

[OK]: REM.1.1

[OK]: REM.2.1

[OK]: REM.3.1

[OK]: REM.3.2

[OK]: REM.4.1

[OK]: REM.4.2

Results of requirements-based testing

[OK]: REM.1.1

[OK]: REM.2.1

[OK]: REM.3.1

[OK]: REM.3.2

[OK]: REM.4.1

[OK]: REM.4.2

Object-level coverage analysis

```
Coverage level: insn
support_rem__my_rem +: 0200057c-0200059b
0200057c +:  2f 84 ff ff      cmpwi   cr7,r4,-1
02000580 +:  41 9e 00 14      beq     cr7,support_rem__my_rem
02000584 +:  7d 23 23 d6      divw    r9,r3,r4
02000588 +:  7c 89 21 d6      mullw   r4,r9,r4
0200058c +:  7c 64 18 50      subf    r3,r4,r3
02000590 +:  4e 80 00 20      blr
02000594 +:  38 60 00 00      li      r3,0
02000598 +:  4e 80 00 20      blr
```

Additional verification result

The untraceable code is functionally correct and corresponds to its requirements. All the requirements were tested and verified, and the instruction coverage analysis showed that all the untraceable object code generated by the compiler was exercised.

5.16 Modulus operator

Test procedure test_mod

Description

Requirements for `x MOD y`. Note that `MOD` is not defined if y = 0.

Requirements and test cases

- Requirement MOD.1: When x >= 0 & y > 0 then x mod y = x - (x/y)*y
  - Test case MOD.1.1: Use two positive operands and verify that the result is also positive.
- Requirement MOD.2: When y = -1, forall x then x mod y = 0 (avoids overflow when x = int'first)
  - Test case MOD.2.1: Use a positive left operand and -1 as the right operand. The expected result of this operation is 0.
  - Test case MOD.2.2: Verify that there is no overflow when the left operand is Integer'First and the right operand is -1. The expected result of this operation is 0.
- Requirement MOD.3: When x >= 0 & y < -1 then x mod y = y - (x mod -y)
  - Test case MOD.3.1: Use a positive left operand and a negative right operand (different from -1). The expected result is negative and corresponds to the difference between the right operand and the result of the mod operation changing the sign of this right operand.
- Requirement MOD.4: When x < 0 & y > 0 then x mod y = y -(-x mod y)
  - Test case MOD.4.1: Use a negative left operand and a positive right operand. The expected result is positive and corresponds to the difference between the right operand and the result of the mod operation changing the sign of the left operand.
- Requirement MOD.5: When x <= 0 and y < -1 then x mod y = -(-x mod -y)
  - Test case MOD.5.1: Use two negative operands. The expected result is negative, and its absolute value corresponds to the mod operation applied to the absolute values of the operands.

Test source code

```
--  Subprogram under test
function My_Mod (A, B : Integer) return Integer is
begin
   return A mod B;
end My_Mod;

--  Test procedure

Val1 : Integer := 83;
Val2 : Integer := 13;
Res1 : Integer := Val1 - (Val1 / Val2) * Val2;
Res2 : Integer := Val2 - Res1;


Assert (My_Mod (Val1,         Val2) =  Res1, "MOD.1.1");
Assert (My_Mod (Val1,           -1) =     0, "MOD.2.1");
Assert (My_Mod (Integer'First, -1) =     0, "MOD.2.2");
Assert (My_Mod (Val1,        -Val2) = -Res2, "MOD.3.1");
Assert (My_Mod (-Val1,        Val2) =  Res2, "MOD.4.1");
Assert (My_Mod (-Val1,       -Val2) = -Res1, "MOD.5.1");
```

Expected results

[OK]: MOD.1.1

[OK]: MOD.2.1

[OK]: MOD.2.2

[OK]: MOD.3.1

[OK]: MOD.4.1

[OK]: MOD.5.1

Results of requirements-based testing

[OK]: MOD.1.1

[OK]: MOD.2.1

[OK]: MOD.2.2

[OK]: MOD.3.1

[OK]: MOD.4.1

[OK]: MOD.5.1

Object-level coverage analysis

```
Coverage level: insn
support_mod__my_mod +: 02000524-0200057b
02000524 +:  2f 84 ff ff      cmpwi   cr7,r4,-1
02000528 +:  41 9e 00 4c      beq     cr7,support_mod__my_mod
0200052c +:  2f 84 00 00      cmpwi   cr7,r4,0
02000530 +:  41 9c 00 1c      blt     cr7,support_mod__my_mod
02000534 +:  2f 83 00 00      cmpwi   cr7,r3,0
02000538 +:  41 9c 00 0c      blt     cr7,support_mod__my_mod
0200053c +:  7d 23 23 d6      divw    r9,r3,r4
02000540 +:  48 00 00 28      b       support_mod__my_mod
02000544 +:  39 23 00 01      addi    r9,r3,1
02000548 +:  48 00 00 18      b       support_mod__my_mod
0200054c +:  39 23 ff ff      addi    r9,r3,-1
02000550 +:  2f 83 00 00      cmpwi   cr7,r3,0
02000554 +:  41 bd 00 0c      bgt     cr7,support_mod__my_mod
02000558 +:  7d 23 23 d6      divw    r9,r3,r4
0200055c +:  48 00 00 0c      b       support_mod__my_mod
02000560 +:  7d 29 23 d6      divw    r9,r9,r4
02000564 +:  39 29 ff ff      addi    r9,r9,-1
02000568 +:  7c 89 21 d6      mullw   r4,r9,r4
0200056c +:  7c 64 18 50      subf    r3,r4,r3
02000570 +:  4e 80 00 20      blr
02000574 +:  38 60 00 00      li      r3,0
02000578 +:  4e 80 00 20      blr
```

Additional verification result

The untraceable code is functionally correct and corresponds to its requirements. All the requirements were tested and verified, and the instruction coverage analysis showed that all the untraceable object code generated by the compiler was exercised.

5.34 Unsigned shift operations

Test procedure test_shifts

Description

Requirements for the left-shift and right-shift operations on unsigned types.

Requirements and test cases

- Requirement SHIFT_LEFT.1: When Shift_Amount <= 7, Shift_Left (X, Shift_Amount) = X * 2**Shift_Amount
  - Test case SHIFT_LEFT.1.1: Verify that the result of Shift_Left is equal to the left operand shifted left by Shift_Amount number of bits when the shift amount is less than or equal to the type's size minus one.
- Requirement SHIFT_LEFT.2: When Shift_Amount > 7 then Shift_Left (X, Shift_Amount) = 0
  - Test case SHIFT_LEFT.2.1: Verify that the result of Shift_Left is equal to zero when the shift amount is greater than or equal to the type's size.
- Requirement SHIFT_RIGHT.1: When Shift_Amount <= 7, Shift_Right (X, Shift_Amount) = X / 2**Shift_Amount
  - Test case SHIFT_RIGHT.1.1: Verify that the result of Shift_Right is equal to the left operand shifted right by Shift_Amount number of bits when the shift amount is less than or equal to the type's size minus one.
- Requirement SHIFT_RIGHT.2: When Shift_Amount > 7 then Shift_Right (X, Shift_Amount) = 0
  - Test case SHIFT_RIGHT.2.1: Verify that the result of Shift_Right is equal to zero when the shift amount is greater than or equal to the type's size.

Test source code

```
--  Subprograms under test
function My_Shift_Left
  (Value : Mod8; Amount : Natural) return Mod8
is
begin
   return
     Mod8 (Shift_Left (Interfaces.Unsigned_8 (Value), Amount));
end My_Shift_Left;

function My_Shift_Right
  (Value : Mod8; Amount : Natural) return Mod8
is
begin
   return
     Mod8 (Shift_Right (Interfaces.Unsigned_8 (Value), Amount));
end My_Shift_Right;

--  Test procedure

Assert (My_Shift_Left (5, 1) = 10
          and then
        My_Shift_Left (7, 2) = 28
          and then
        My_Shift_Left (1, 7) = 128,
        "SHIFT_LEFT.1.1");
Assert (My_Shift_Left (123, 8) = 0
          and then
        My_Shift_Left (1, 10) = 0,
        "SHIFT_LEFT.2.1");
```

```
Assert (My_Shift_Right (5, 1) = 2
          and then
        My_Shift_Right (7, 2) = 1
          and then
        My_Shift_Right (255, 7) = 1,
        "SHIFT_RIGHT.1.1");
Assert (My_Shift_Right (123, 8) = 0
          and then
        My_Shift_Right (255, 10) = 0,
        "SHIFT_RIGHT.2.1");
```

Expected results

[OK]: SHIFT_LEFT.1.1

[OK]: SHIFT_LEFT.2.1

[OK]: SHIFT_RIGHT.1.1

[OK]: SHIFT_RIGHT.2.1

Results of requirements-based testing

[OK]: SHIFT_LEFT.1.1

[OK]: SHIFT_LEFT.2.1

[OK]: SHIFT_RIGHT.1.1

[OK]: SHIFT_RIGHT.2.1

Object-level coverage analysis

```
Coverage level: insn
support_shifts__my_shift_left +: 02000584-0200059f
02000584 +:  2f 84 00 07      cmpwi   cr7,r4,7
02000588 +:  41 9d 00 10      bgt     cr7,support_shifts__my_shift_left
0200058c +:  7c 63 20 30      slw     r3,r3,r4
02000590 +:  54 63 06 3e      clrlwi  r3,r3,24
02000594 +:  4e 80 00 20      blr
02000598 +:  38 60 00 00      li      r3,0
0200059c +:  4e 80 00 20      blr
support_shifts__my_shift_right +: 020005a0-020005bb
020005a0 +:  2f 84 00 07      cmpwi   cr7,r4,7
020005a4 +:  41 9d 00 10      bgt     cr7,support_shifts__my_shift_right
020005a8 +:  7c 63 24 30      srw     r3,r3,r4
020005ac +:  54 63 06 3e      clrlwi  r3,r3,24
020005b0 +:  4e 80 00 20      blr
020005b4 +:  38 60 00 00      li      r3,0
020005b8 +:  4e 80 00 20      blr
```

Additional verification result

The untraceable code is functionally correct and corresponds to its requirements. All the requirements were tested and verified, and the instruction coverage analysis showed that all the untraceable object code generated by the compiler was exercised.

CHAPTER

SIX

CONCLUSION

Chapter *Introduction* provides a set of compilation options and restrictions that minimize source code to object code traceability issues. They must be enforced by the applicant, as well as the identified compiler version, in order to ensure that its application is covered by this study.

Chapters *Ada Features Survey* and *Combined Ada Features Survey* provide a comprehensive analysis, with 344 compilable code samples, including 11 specific samples addressing combinations of elementary Ada features. No new traceability issue was detected by making the source code more complex through combinations of elementary language features. The generated object code was carefully analyzed to verify its mapping to the original source code, and the results of this inspection were consolidated in this document.

From these code samples, several occurrences of untraceable code were detected. 48 additional verifications have been performed (see chapter *Additional Verifications*), comprising the following items:

1. specification of the requirements describing the intended behavior of the untraceable code,
2. definition of the test cases and procedures to verify the compliance against the requirements,
3. execution and collection of test results and coverage information, and
4. verification of test results and assessment of object code coverage.

Requirements-based testing for the occurrences of untraceable code was successfully achieved with no deviations from the expected behavior.

There is a case where the compiler generates different patterns of code depending on the ranges of array aggregates in use. Section *Index Constraints and Discrete Ranges* studies array initialization, for which the compiler generates either a static initializer or run-time code, and both possibilities where analyzed in depth. The code generation strategy for case statements is also dependent on the values in the case statement alternatives, as analyzed in section *Case Statements*.

Ada constructs were thoroughly analyzed, and every occurrence of untraceable object code was verified and assessed to be correct. Hence, it can be concluded from this analysis that for all the Ada features mentioned in chapter *Ada Features Survey*, when using the options, and restrictions described in chapter *Introduction*, the compiler GNAT Pro High-Integrity Edition 7.4.3 generates object code which is directly traceable to the source code except for the specific cases mentioned above. For these specific cases, additional analysis has been performed to ensure that the additional object code is functionally correct.

As a result of these analyses, the correlation of source code to object code within the scope of the study is fully understood and justified.

APPENDIX

A

EXTERNAL CALLS INDEX

This chapter defines the set of calls to operating system or user-defined routines that may be generated by the compiler, together with the locations where these calls appear in the study. Certification material would be required for these routines.

A.1 External Calls Definitions

The external calls that may be generated by the compiler can be grouped into two categories: operating system and user-defined routines. Their definition is the following:

- Last Chance Handler. When an exception is raised, a call is made to the user-defined routine designated by the symbol `__gnat_last_chance_handler`. All last chance handler implementations must terminate or suspend the thread that executes the handler. This handler can be written in C or in Ada:

```
procedure Last_Chance_Handler
  (Msg  : System.Address; Line : Integer);
pragma Export
  (C, Last_Chance_Handler,
   "__gnat_last_chance_handler");
pragma No_Return (Last_Chance_Handler);
```

- Allocator. For runtime library profiles where dynamlic memory allocation is supported, dynamic memory allocations (heap) generate calls to a C convention user-defined function which returns the address of a memory block with the requested size. The corresponding Ada subprogram declaration is:

```
function Gnat_Malloc
  (size : Interfaces.C.size_t) return System.Address;
pragma Export (C, Gnat_Malloc, "__gnat_malloc");
```

- Memory block comparison. When comparing objects, the compiler may generate calls to the ISO C standard `memcmp` function which is provided by the underlying operating system. The corresponding Ada and C subprogram declaration is:

```
function Memcmp (S1 : Address; S2 : Address; N : size_t) return int;
pragma Export (C, Memcmp, "memcmp");

int memcmp(const void *s1, const void *s2, size_t n);
```

- Memory block copy. The compiler may generate calls to the ISO C standard `memcpy` function, provided by the underlying operating system, to implement assignment statements efficiently when memory areas do not overlap. The corresponding Ada and C subprogram declaration is:

```
function Memcpy (Dest : Address; Src : Address; N : size_t) return Address;
pragma Export (C, Memcpy, "memcpy");

void *memcpy(void *dest, const void *src, size_t n);
```

- Memory block copy with possible overlap. The compiler may generate calls to the ISO C standard `memmove` function, provided by the underlying operating system, to implement assignment statements efficiently when memory areas may overlap. The corresponding Ada and C subprogram declaration is:

```
function Memmove
  (Dest : Address; Src : Address; N : size_t) return Address;
pragma Export (C, Memmove, "memmove");
```

```
void *memmove(void *dest, const void *src, size_t n);
```

- Memory block initialization. The ISO C standard `memset` function, provided by the underlying operating system, can be generated by the compiler to perform efficient object initializations. The corresponding Ada and C subprogram declaration is:

```
function Memset (S : Address; C : int; N : size_t) return Address;
pragma Export (C, Memset, "memset");
```

```
void *memset(void *s, int c, size_t n);
```

- Floating point value validity check. The implementation of 'Valid for floating point types relies on a supporting generic function which is instantiated for each base floating point type. The corresponding Ada and C subprogram declarations are:

```
function Valid (X : not null access T) return Boolean;
--  Where T is Short_Float, Float, Long_Float, or Long_Long_Float
```

```
unsigned char system__fat_XXX__attr_YYY__valid (T *x);
/*  Where T is float, float, double, or long double,
    XXX is sflt, flt, lflt, or llf,
    and YYY is short_float, float, long_float, or long_long_float.  */
```

- Attributes of floating point types. The function attributes of floating point types are provided as instances of a set of generic functions which is instantiated for each base floating point type. The corresponding Ada and C subprogram declarations are:

```
subtype UI is Integer;

function Adjacent          (X, Towards : T)              return T;
function Ceiling           (X : T)                       return T;
function Compose           (Fraction : T; Exponent : UI) return T;
function Copy_Sign         (Value, Sign : T)             return T;
function Exponent          (X : T)                       return UI;
function Floor             (X : T)                       return T;
function Fraction          (X : T)                       return T;
function Leading_Part      (X : T; Radix_Digits : UI)    return T;
function Machine           (X : T)                       return T;
function Machine_Rounding  (X : T)                       return T;
function Model             (X : T)                       return T;
function Pred              (X : T)                       return T;
function Remainder         (X, Y : T)                    return T;
function Rounding          (X : T)                       return T;
function Scaling           (X : T; Adjustment : UI)      return T;
function Succ              (X : T)                       return T;
function Truncation        (X : T)                       return T;
function Unbiased_Rounding (X : T)                       return T;
--  Where T is Short_Float, Float, Long_Float, or Long_Long_Float
```

```
T system__fat_XXX__attr_YYY__adjacent (T, T);
T system__fat_XXX__attr_YYY__ceiling (T);
T system__fat_XXX__attr_YYY__compose (T, int);
T system__fat_XXX__attr_YYY__copy_sign (T, T);
T system__fat_XXX__attr_YYY__exponent (T);
T system__fat_XXX__attr_YYY__floor (T);
```

```
T system__fat_XXX__attr_YYY__fraction (T);
T system__fat_XXX__attr_YYY__leading_part (T, int);
T system__fat_XXX__attr_YYY__machine (T);
T system__fat_XXX__attr_YYY__machine_rounding (T);
T system__fat_XXX__attr_YYY__model (T);
T system__fat_XXX__attr_YYY__pred (T);
T system__fat_XXX__attr_YYY__remainder (T, T);
T system__fat_XXX__attr_YYY__rounding (T);
T system__fat_XXX__attr_YYY__scaling (T, int);
T system__fat_XXX__attr_YYY__succ (T);
T system__fat_XXX__attr_YYY__truncation (T);
T system__fat_XXX__attr_YYY__unbiased_rounding (T);
/*  Where T is float, float, double, or long double,
    XXX is sflt, flt, lflt, or llf,
    and YYY is short_float, float, long_float, or long_long_float.  */
```

A.2 External Calls References

- __gnat_last_chance_handler (*ch110300_01*, *ch110300_02*, *ch130701_02*)
- __gnat_malloc (*ch030601_06_dynalloc*, *ch030603_02_dynalloc*, *ch040800_01*, *ch040800_02*, *ch040800_04*, *ch040800_05*, *ch040800_06*, *ch040800_07*, *ch040800_08*, *ch040800_09*)
- memcmp (*ch030900_03*, *ch040502_02*)
- memcpy (*ch030900_03*, *ch040301_06*, *ch040700_03*, *ch040700_04*, *ch040800_05*, *ch050200_03*, *ch050200_05*, *ch050200_07*, *ch060400_04*)
- memmove (*ch040102_02*, *ch050200_06*)
- memset (*ch030603_02_dynalloc*)
- system__fat_flt__attr_float__valid (*ch130902_02*)
- system__fat_sflt__attr_long_float__adjacent (*ch0a0503_01*)
- system__fat_sflt__attr_long_float__ceiling (*ch0a0503_01*)
- system__fat_sflt__attr_long_float__compose (*ch0a0503_01*)
- system__fat_sflt__attr_long_float__copy_sign (*ch0a0503_01*)
- system__fat_sflt__attr_long_float__exponent (*ch0a0503_01*)
- system__fat_sflt__attr_long_float__floor (*ch0a0503_01*)
- system__fat_sflt__attr_long_float__fraction (*ch0a0503_01*)
- system__fat_sflt__attr_long_float__leading_part (*ch0a0503_01*)
- system__fat_sflt__attr_long_float__machine_rounding (*ch0a0503_01*)
- system__fat_sflt__attr_long_float__remainder (*ch0a0503_01*)
- system__fat_sflt__attr_long_float__rounding (*ch0a0503_01*)
- system__fat_sflt__attr_long_float__scaling (*ch0a0503_01*)
- system__fat_sflt__attr_long_float__truncation (*ch0a0503_01*)
- system__fat_sflt__attr_long_float__unbiased_rounding (*ch0a0503_01*)
- system__secondary_stack__ss_allocate (*ch030700_02_sec_stack*, *ch030900_01a*)
- system__secondary_stack__ss_mark (*ch030700_02_sec_stack*, *ch030902_01*, *ch030902_02a*, *ch030902_02b*, *ch030902_05*, *ch030902_06*, *ch030903_02a*, *ch030903_03a*, *ch030904_01a*, *ch030904_02a*, *ch030904_03a*)