AUnit Cookbook

AUnit - version 2.01
Document revision level $Revision: 1.15 $
Date: 5 June 2007

AdaCore

http://www.adacore.com

Table of Contents

1 Introduction 1

2 Overview 3

3 Simple Test Case 5

4 Fixture 7

5 Suite 9

6 Composition of Suites 11

7 Support for OOP 13

8 Support for Generics 17

9 Reporting 19

10 Migrating Tests from AUnit 1 to AUnit 2 21

11 Using AUnit with Restricted Run-Time Libraries 23

12 Installation and Use 25

12.1 Installing AUnit on UNIX systems 25
12.2 Installing AUnit on Windows systems 25
12.3 Installed files 26

13 GPS Support 27

1 Introduction

This is a short guide for using the AUnit test framework. AUnit is an adaptation of the Java JUnit (Kent Beck, Erich Gamma) unit test framework for Ada code. AUnit 2 differs somewhat from the original AUnit 1 in that it is compatible with restricted run-time libraries provided with GNAT Pro for high integrity applications. It also provides better support for testing of OOP applications by allowing overriding and inheritance of test routines. AUnit 1 only supported inheritance.

AUnit allows a great deal of flexibility as to the structure of test cases, suites and harnesses. The templates and examples given in this document illustrate how to use AUnit while staying within the constraints of the GNAT Pro restricted and Zero Foot Print (ZFP) run-time libraries. Therefore, they avoid the use of dynamic allocation and some other features that would be outside of the profiles corresponding to these libraries. Tests targeted to the full Ada run-time library need not comply with these constraints.

This document is adapted from the JUnit Cookbook document contained in the JUnit release package.

Special thanks to Francois Brun of Thales Avionics for his ideas about support for OOP testing.

2 Overview

How do you write testing code?

The simplest way is as an expression in a debugger. You can change debug expressions without recompiling, and you can wait to decide what to write until you have seen the running objects. You can also write test expressions as statements that print to the standard output stream. Both styles of tests are limited because they require human judgment to analyze their results. Also, they don't compose nicely - you can only execute one debug expression at a time and a program with too many print statements causes the dreaded "Scroll Blindness".

AUnit tests do not require human judgment to interpret, and it is easy to run many of them at the same time. When you need to test something, here is what you do:

1. Instantiate `AUnit_Framework.Framework` to parameterize the test framework. Since restricted run-time profiles do not support dynamic memory management without customization, tables of information used for reporting test results and for manipulating tests must be statically sized. In this document, we will call this instantiation **AUnit** in both code samples and text. When referring to this user-instantiated package, it will appear in boldface, as will all other user-written code.
2. Declare a package for a test case - a set of logically related test routines.
3. Derive a test case type from **AUnit**`.Test_Cases.Test_Case` in the package.
4. The new derived type must provide implementations of `Register_Tests` and `Name`.
5. Write each test routine (see below) and register it with a statement in routine `Register_Tests`, of the form:

   ```
   Register_Routine (T, Test_Name'Access, "Description of test routine");
   ```

   `Register_Routine` is exported by **AUnit**`.Test_Cases.Registration`.
6. When you want to check a value, use:

   ```
   AUnit.Test_Cases.Assertions.Assert (Boolean_Expression, String_Description);
   ```

   or:

   ```
   if not AUnit.Test_Cases.Assertions.Assert (Boolean_Expression, String_Description) then
      return;
   end if;
   ```

   Note that in AUnit 2 the procedural form of `Assert` will have different behavior depending on whether the underlying Ada run-time library supports exception handling. If exception handling is supported, a failed assertion will cause the execution of the calling test routine to be abandoned. If exception handling is not supported, a failed assertion will not do this: the calling test routine will continue executing. This behavior is selected by the `SUPPORT_EXCEPTION` variable in the *makefile* file, during AUnit compilation. The first behavior is identical to that of AUnit 1 for backward compatibility. The functional form of `Assert` always continues on a failed assertion, and provides you with a choice of behaviors. This form allows writing test routines that are fully portable across run-time profiles.
7. Create a suite function inside a package to gather together test cases and sub-suites. Test cases and suites must be statically allocated if using the ZFP profile without custom dynamic memory management, or the "cert" run-time profile.
8. At any level at which you wish to run tests, create a harness by instantiating procedure **AUnit**`.Test_Runner` or function **AUnit**`.Test_Runner_With_Status` with the top-level suite

function to be executed. This instantiation provides a routine that executes all of the tests in the suite. We will call this user-instantiated routine **Run** in the text for backward compatibility to tests developed for AUnit 1. Note that only one instance of **Run** can execute at a time. This is a tradeoff made to reduce the stack requirement of the framework by allocating test result reporting data structures statically. **AUnit**.`Test_Runner` must be instantiated at the library level.

9. Build the code that calls the harness **Run** routine using gnatmake. The GNAT project file *aunit.gpr* contains all necessary switches, and should be imported into your root project file.

The first step when using AUnit is to instantiate the framework in order to size various elements used for manipulating tests and reporting the results of a run. Unlike AUnit 1, AUnit 2 does not use any dynamic allocation so that it can be used with GNAT run-time libraries such as ZFP and cert that either do not support the use of allocators by default, or place restrictions on where they can be used. The following code fragment is a typical instantiation. User-specific text is in boldface.

```
with AUnit_Framework.Framework;
package AUnit is new AUnit_Framework.Framework
  (Max_Exceptions_Per_Harness   => 5,
   Max_Failures_Per_Harness     => 20,
   Max_Routines_Per_Test_Case   => 50,
   Max_Test_Cases_Per_Suite     => 50,
   Message_String_Pool_Size     => 4096;
```

The term "run" in this discussion means the execution of a call to the **Run** routine that is an instantiation of either **AUnit**.`Test_Runner` or **AUnit**.`Test_Runner_With_Result`.

The formal parameters of `AUnit_Framework.Framework` are:

- *Max_Exceptions_Per_Harness*: the maximum number of unhandled exceptions that can be reported in test routines for a given run. If this value is exceeded, a warning is displayed and the exception is reported immediately rather than as part of the summary reporting.
- *Max_Failures_Per_Harness*: the maximum number of failed assertions that can be reported in the run.. If this value is exceeded, a warning is displayed and the exception is reported immediately rather than as part of the summary reporting.
- *Max_Routines_Per_Test_Case*: the maximum number of routines that can be part of any test case. If this limit is exceeded, a warning is displayed and the routine in question is not added to the test case for execution.
- *Max_Test_Cases_Per_Suite*: the maximum number of test cases or subsuites that can be put into a suite. If this limit is exceeded, a warning is displayed and the test case is not added to the suite for execution.
- *Message_String_Pool__Size*: amount of space in bytes to be statically allocated for strings used in the AUnit report. Default value is 10_000.

3 Simple Test Case

To test that the sum of two Moneys with the same currency contains a value which is the sum of the values of the two Moneys, the test routine would look like:

```
procedure Test_Simple_Add (T : in out Money_Test) is
    X, Y: Some_Currency;
begin
   X := 12; Y := 14;
   Assert (X + Y = 26, "Addition is incorrect");
end Test_Simple_Add;
```

The package spec looks as follows. The only modification was to remove support for a test fixture (next section), and to provide a name for the unit. Changes to "boilerplate code" are in bold (remember that **AUnit** here is the name of your instantiation of `AUnit_Framework.Framework`).

```
with AUnit; use AUnit;

package Money_Tests is
   use Test_Results;

   type Money_Test is new Test_Cases.Test_Case with null record;

   procedure Register_Tests (T: in out Money_Test);
   -- Register routines to be run

   function Name (T: Money_Test) return Test_String;
   -- Provide name identifying the test case

   -- Test Routines:
   procedure Test_Simple_Add (T : in out Test_Cases.Test_Case'Class);
end Money_Tests;
```

The package body is:

```
package body Money_Tests is

   use Assertions;

   procedure Test_Simple_Add (T : in out Test_Cases.Test_Case'Class) is
        X, Y : Some_Currency;
   begin
      X := 12; Y := 14;
      Assert (X + Y = 26, "Addition is incorrect");
   end Test_Simple_Add;

   -- Register test routines to call
   procedure Register_Tests (T: in out Money_Test) is

      use Test_Cases.Registration;

   begin
      -- Repeat for each test routine:
      Register_Routine (T, Test_Simple_Add'Access, "Test Addition");
   end Register_Tests;

   -- Identifier of test case

   function Name (T: Money_Test) return Test_String is
   begin
      return Format ("Money Tests");
   end Name;

end Money_Tests;
```

The corresponding harness code, which imports user suite package **Money_Suite** (see below) is:

```
with Money_Suite;
with AUnit;
procedure Run is new AUnit.Test_Runner (Money_Suite.Suite);

with Run;
--  If targeting the ZFP run-time library, uncomment:
--  with Last_Chance_Handler, Dummy_SS_Get;
procedure My_Tests is
begin
   Run;
end My_Tests;
```

4 Fixture

Tests need to run against the background of a set of known entities. This set is called a test fixture. When you are writing tests you will often find that you spend more time writing code to set up the fixture than you do in actually testing values.

You can make writing fixture code easier by sharing it. Often you will be able to use the same fixture for several different tests. Each case will send slightly different messages or parameters to the fixture and will check for different results.

When you have a common fixture, here is what you do:

1. Create a package as in the previous section.
2. Declare variables or components for elements of the fixture either as part of the test case type or in the package body.
3. Override `Set_Up_Case` to initialize the fixture for all test routines.
4. Override `Set_Up` to initialize the variables or components before the execution of each routine.
5. Override `Tear_Down` to release any resources you allocated in `Set_Up` - to be executed after each test routine.
6. Override `Tear_Down_Case` to release any permanent resources you allocated in `Set_Up_Case` - to be executed after all test routines.

For example, to write several test cases that want to work with different combinations of 12 Euros, 14 Euros, and 26 US Dollars, first create a fixture. The package spec is now:

```
with AUnit; use AUnit;

package Money_Tests is
   use Test_Results;

   type Money_Test is new Test_Cases.Test_Case with null record;

   procedure Register_Tests (T: in out Money_Test);
   -- Register routines to be run

   function Name (T : Money_Test) return Test_String;
   -- Provide name identifying the test case

   procedure Set_Up (T : in out Money_Test);
   --  Set up performed before each test routine

   -- Test Routines:
   procedure Test_Simple_Add (T : in out Test_Cases.Test_Case'Class);
end Money_Tests;
```

The body becomes:

```
package body Money_Tests is

   use Assertions;

   -- Fixture elements

   EU_12, EU_14 : Euro;
  US_26       : US_Dollar;

   -- Preparation performed before each routine

   procedure Set_Up (T: in out Money_Test) is
   begin
      EU_12 := 12; EU_14 := 14;
    US_26 := 26;
   end Set_Up;

   procedure Test_Simple_Add (T : in out Test_Cases.Test_Case'Class) is
        X, Y : Some_Currency;
   begin
        Assert
      (EU_12 + EU_14 /= US_26,
       "US and EU currencies not differentiated");
   end Test_Simple_Add;

   -- Register test routines to call
   procedure Register_Tests (T: in out Money_Test) is

      use Test_Cases.Registration;

   begin
      -- Repeat for each test routine:
      Register_Routine (T, Test_Simple_Add'Access, "Test Addition");
   end Register_Tests;

   -- Identifier of test case
   function Name (T: Money_Test) return Test_String is
   begin
      return Format ("Money Tests");
   end Name;

end Money_Tests;
```

Once you have the fixture in place, you can write as many test routines as you like. Calls to `Set_Up` and `Tear_Down` bracket the invocation of each test routine.

Once you have several test cases, organize them into a Suite.

5 Suite

How do you run several test cases at once?

As soon as you have two tests, you'll want to run them together. You could run the tests one at a time yourself, but you would quickly grow tired of that. Instead, AUnit provides an object, `Test_Suite`, that runs any number of test cases together.

For test routines that use the same fixture (i.e. those declared in the same package), the `Register_Routine` procedure is used to collect them into the single test case.

To create a suite of two test cases and run them together, first create a test suite (again, **AUnit** designates the user instantiation of `AUnit_Framework.Framework`):

```
with AUnit; use AUnit;
package My_Suite is
   function Suite return Test_Suites.Access_Test_Suite;
end My_Suite;

--  Import tests and sub-suites to run
with Test_Case_1, Test_Case_2;

package body My_Suite is
   use Test_Suites;

   -- Statically allocate test suite:
   Result : aliased Test_Suite;

   --  Statically allocate test cases:
   Test_1 : aliased Test_Case_1.Test_Case;
   Test_2 : aliased Test_Case_2.Test_Case;

   function Suite return Acces_Test_Suite is
   begin
      Add_Test (Result'Access, Test_Case_1'Access);
      Add_Test (Result'Access, Test_Case_2'Access);
      return Result'Access;
   end Suite;
end My_Suite;
```

The harness and test procedure are:

```
with My_Suite;
with AUnit;
procedure Run is new AUnit.Test_Runner (My_Suite.Suite);

with Run;
--  If targeting the ZFP run-time library, uncomment:
--  with Last_Chance_Handler, Dummy_SS_Get;
procedure My_Tests is
begin
   Run;
end My_Tests;
```

6 Composition of Suites

Typically, one will want the flexibility to execute a complete set of tests, or some subset of them. In order to facilitate this, we can compose both suites and test cases, and provide a harness for any given suite:

```
-- Composition package:
with AUnit; use AUnit;
package Composite_Suite is
   function Suite return Test_Suites.Access_Test_Suite;
end Composite_Suite;

--  Import tests and suites to run
with This_Suite, That_Suite;

package body Composite_Suite is
   use Test_Suites;

   -- Statically allocate test suite. Note that the suites to compose
   -- have already been allocated in their own packages.
   Result : aliased Test_Suite;

   function Suite return Access_Test_Suite is
   begin
      Add_Test (Result'Access, This_Suite.Suite'Access);
      Add_Test (Result'Access, That_Suite.Suite'Access);
      return Result'Access;
   end Suite;
end Composite_Suite;
```

The harness remains the same:

```
with Composite_Suite;
with AUnit;
procedure Run is new AUnit.Test_Runner (Composite_Suite.Suite);

with Run;
--  If targeting the ZFP run-time library, uncomment:
--  with Last_Chance_Handler, Dummy_SS_Get;
procedure Composite_Tests is
begin
   Run;
end Composite_Tests;
```

As can be seen, this is a very flexible way of composing test cases into execution runs: any combination of test cases and sub-suites can be collected into a suite.

7 Support for OOP

When testing a hierarchy of tagged types, one will often want to run tests for parent types against their derivations without rewriting those tests. The most straightforward way to accomplish this is to derive the test cases for the derived types from those for the parent type.

Suppose we have a parent type defined in a package:

```
package Root is
   type Parent is tagged private;

   procedure P1 (P : in out Parent);
   procedure P2 (P : in out Parent);
private
   type Parent is tagged null record;
end Root;
```

and a corresponding test case:

```
with AUnit; use AUnit;
package Parent_Tests is
   use Test_Results;

   type Parent_Test is new Test_Cases.Test_Case with private;

   function Name (P : Parent_Test) return Test_String;
   procedure Register_Tests (P : in out Parent_Test);

   --  Test routines. If derived types are declared in child packages,
   --  these can be in the private part.
   procedure Test_P1 (P : in out Test_Cases.Test_Case'Class);
   procedure Test_P2 (P : in out Test_Cases.Test_Case'Class);

private
   type Parent_Test is new Test_Cases.Test_Case with null record;
end Parent_Tests;
```

The body of the test case will follow the usual pattern, declaring one or more objects of type **Parent**, and executing statements in the test routines against them. However, in order to support dispatching to overriding routines of derived test cases, we need to introduce class-wide wrapper routines for each primitive test routine of the parent type that we anticipate may be overridden. Instead of registering the parent's overridable primitive operations directly using `Register_Routine`, we register the wrapper using `Register_Wrapper`. This latter routine is exported by instantiating **AUnit**`.Test_Cases.Specific_Test_Case_Registration` with the actual parameter being the parent test case type.

```
with Root; use Root;
package body Parent_Tests is
   use Assertions;

   Fixture : Parent;
   --  This could also be a field of Parent_Test

   --  Declare class-wide wrapper routines for any test routines that will be
   --  overridden:
   procedure Test_P1_Wrapper (P : in out Parent_Test'Class);
   procedure Test_P2_Wrapper (P : in out Parent_Test'Class);

   function Name (C : Parent_Test) return Test_String is ...;

   --  Register Wrappers:
   procedure Register_Tests (P : in out Parent_Test) is

      package Register_Specific is
        new Test_Cases.Specific_Test_Case_Registration (Parent_Test);

      use Register_Specific;

   begin
      Register_Wrapper (P, Test_P1_Wrapper'Access, "Test P1");
      Register_Wrapper (P, Test_P2_Wrapper'Access, "Test P2");
   end Register_Tests;

   --  Test routines:
   procedure Test_P1 (P : in out Test_Cases.Test_Case'Class) is ...;
   procedure Test_P2 (C : in out Test_Cases.Test_Case'Class) is ...;

   --  Wrapper routines. These dispatch to the corresponding primitive
   --  test routines of the specific types.
   procedure Test_P1_Wrapper (P : in out Parent_Test'Class) is
   begin
      Test_P1 (P);
   end Test_P1_Wrapper;

   procedure Test_P2_Wrapper (P : in out Parent_Test'Class) is
   begin
      Test_P2 (P);
   end Test_P2_Wrapper;

end Parent_Tests;
```

Now consider a derivation of type `Parent`:

```
with Root;
package Branch is
   type Child is new Root.Parent with private;

   procedure P2 (C : in out Child);
   procedure P3 (C : in out Child);
private
   type Child is new Root.Parent with null record;
end Branch;
```

Note that `Child` retains the parent implementation of `P1`, overrides `P2` and adds `P3`. Its test case looks like the following, assuming that we will override `Test_P2` when we override `P2` (not necessary, but certainly possible):

```
with Parent_Tests; use Parent_Tests;
with AUnit; use AUnit;
package Child_Tests is
   use Test_Results;

   type Child_Test is new Parent_Test with private;

   function Name (C : Child_Test) return Test_String;
   procedure Register_Tests (C : in out Child_Test);

   --  Test routines:
   procedure Test_P2 (C : in out Test_Cases.Test_Case'Class);
   procedure Test_P3 (C : in out Test_Cases.Test_Case'Class);

private
   type Child_Test is new Parent_Test with null record;
end Child_Tests;

with Branch; use Branch;
package body Child_Tests is
   use Assertions;

   Fixture : Child;
   --  This could also be a field of Child_Test

   --  Declare wrapper for Test_P3:
   procedure Test_P3_Wrapper (C : in out Child_Test'Class);

   function Name (C : Child_Test) return Test_String is ...;

   procedure Register_Tests (C : in out Child_Test) is

      package Register_Specific is
         new Test_Cases.Specific_Test_Case_Registration (Child_Test);
      use Register_Specific;

   begin
      -- Register parent tests for P1 and P2:
      Parent_Tests.Register_Tests (Parent_Test (C));

      -- Repeat for each new test routine (Test_P3 in this case):
      Register_Wrapper (C, Test_P3_Wrapper'Access, "Test P3");
   end Register_Tests;

   --  Test routines:
   procedure Test_P2 (C : in out Test_Cases.Test_Case'Class) is ...;
   procedure Test_P3 (C : in out Test_Cases.Test_Case'Class) is ...;

   --  Wrapper for new routine:
   procedure Test_P3_Wrapper (C : in out Child_Test'Class) is
   begin
      Test_P3 (C);
   end Test_P3_Wrapper;

end Child_Tests;
```

Note that inherited and overridden tests do not need to be explicitly re-registered in derived test cases - one just calls the parent version of `Register_Tests`. If the application tagged type hierarchy is organized into parent and child units, one could also organize the test cases into a hierarchy that reflects that of the units under test.

8 Support for Generics

When testing generic units, one would like to apply the same generic tests to all instantiations in an application. A simple approach is to make the generic unit under test a formal parameter to a generic test case.

For instance, suppose the generic unit to test is a package (though it could be a subprogram, and the same principle would apply):

```
generic
   -- Formal parameter list
package Template is
   -- Declarations
end Template;
```

The corresponding test case would be:

```
with AUnit; use AUnit;
with Template;
generic
   with package Instance is new Template (<>);
package Template_Tests is
   use Test_Results;

   type Template_Test is new Test_Cases.Test_Case with private;

   function Name (T : Template_Test) return Test_String;
   procedure Register_Tests (T : in out Template_Test);

   --  Declare test routines

private

   type Template_Test is new Test_Cases.Test_Case with ...;

   -- Specifications of test routines, and declarations of access values
   -- to them for use in Register_Routine:

end Template_Tests;
```

The body will follow the usual patterns with the fixture being based on the formal package `Instance`. Note that due to a recent AI, accesses to test routines, along with the test routine specifications, must be defined in the package specification rather than in its body.

Instances of `Template` will have associated instances of `Template_Tests`. The instances under test can be instantiated as is convenient, at the library level or within a helper package. Likewise for the test case instances. The instantiated test case objects are added to a suite in the usual manner.

9 Reporting

Currently test results are reported using a simple console reporting routine that is invoked when the `Run` routine of an instantiation of **AUnit**`.Test_Runner` is called.

Here is an example where the test program calls three different `Run` routines (because they need to be invoked in the contexts of distinct tasks):

```
--------------------

   Total Tests Run: 7

   Successful Tests: 7
      Apex_Blackboards_Init         : creation routines
      Apex_Buffers_Init             : creation routines
      Apex_Events_Init              : creation routines
      Apex_Processes_Init           : creation routines
      Apex_Queuing_Ports_Init       : creation routines
      Apex_Sampling_Ports_Init      : creation routines
      Apex_Semaphores_Init          : creation routines

   Failed Assertions: 0

--------------------

   Total Tests Run: 4

   Successful Tests: 4
      Apex_Partition                : Partition status
      Apex_Processes                : General routines for ARINC processes
      Apex_Processes                : Routines that manipulate process priority
      Apex_Processes                : Routines that stop and resume processes

   Failed Assertions: 0

--------------------

   Total Tests Run: 46

   Successful Tests: 44
      Apex_Semaphores               : Routines that handle semaphores
      Apex_Buffers                  : Routines that handle buffers
      Apex_Blackboards              : Blackboard information exchange
      Apex_Queuing_Ports            : Routines using queuing ports
      Apex_Sampling_Ports           : Routines using sampling ports
      Exception handling            : Local exception handling in APEX process
      Exception handling            : Synch signal handling in APEX process
      Exception handling            : Integer overflow handling in APEX process
      HW FP error detection         : Divide by zero - constrained float
      HW FP error detection         : Divide by zero - unconstrained float
      HW FP error detection         : Constrained floating point overflow
      HW FP error detection         : Unconstrained floating point overflow
      HW FP error detection         : Divide by zero - C4A012B
      HW FP error detection         : Divide by zero - C4A013A
      HW FP error detection         : Overflows on addition and subtraction - C45322A
      HW FP error detection         : Overflows on multiplication and division - C45523A
      HW FP error detection         : Overflows on exponentiation - C45622A
      Stack Overflow                : Overflow on large objects
      Stack Overflow                : Basic overflow handling in APEX process
      Stack Overflow                : Cascaded overflow in exception handler
      Stack Overflow                : Overflow due to large string allocation
      Generic_Elementary_Functions  : CXG2001: accuracy of Standard.Float
      Generic_Elementary_Functions  : CXG2003: accuracy of sqrt
      Generic_Elementary_Functions  : CXG2004: accuracy of sin and cos
      Generic_Elementary_Functions  : CXG2005: accuracy of FP add and multiply
      Generic_Elementary_Functions  : CXG2010: accuracy of exp
      Generic_Elementary_Functions  : CXG2011: accuracy of log
      Generic_Elementary_Functions  : CXG2012: accuracy of **
      Generic_Elementary_Functions  : CXG2013: accuracy of tan/cot: float exact
      Generic_Elementary_Functions  : CXG2013: accuracy of tan: float +/- pi
      Generic_Elementary_Functions  : CXG2013: accuracy of tan: float fractional
      Generic_Elementary_Functions  : CXG2013: accuracy of cot: float
      Generic_Elementary_Functions  : CXG2013: accuracy of tan/cot: float exception
      Generic_Elementary_Functions  : CXG2013: accuracy of tan/cot: long exact
      Generic_Elementary_Functions  : CXG2013: accuracy of tan: long +/- pi
      Generic_Elementary_Functions  : CXG2013: accuracy of tan: long fractional
      Generic_Elementary_Functions  : CXG2013: accuracy of cot: long
      Generic_Elementary_Functions  : CXG2013: accuracy of tan/cot: long exception
      Generic_Elementary_Functions  : CXG2015: accuracy of arcsin and arccos
```

10 Migrating Tests from AUnit 1 to AUnit 2

In adapting AUnit to be usable in a restricted Ada run-time library context, we were faced with a number of issues (e.g. no controlled types, no dynamic allocation, no exception handlers, no unconstrained function results) that inevitably broke backward compatibility between AUnit 1 and AUnit 2. We have tried to minimize these incompatibilities, but nonetheless converting test cases, suites and harnesses designed to work with AUnit 1 requires some effort. The following steps describe the conversion, assuming you are using the same run-time library for both versions of AUnit.

1. Provide an AUnit package

   With AUnit now supporting constrained run-times, all AUnit internal types are constrained. Technically, this means that the AUnit framework is now a generic package whose instantiation will provide the constraints. In order to port your tests from the previous AUnit framework, you need to instantiate the '`AUnit_Framework.Framework`' package. By naming this instantiation `AUnit`, you will minimize the steps to achieve the conversion of old tests. The instantiation and its parameters is described in the Overview section.

2. Modify context clauses

   All of the previous `AUnit.xxx` packages are now declared in `AUnit_Framework.Framework`. Therefore, you don't need to (and can't) 'with' them directly in your code. For example, `with AUnit.Test_Suites; use AUnit.Test_Suites;` will now result in a compilation error. You should replace the above example with `with` **`AUnit`**`; use` **`AUnit`**`.Test_Suites;`

3. API modifications

   The `Test_Cases` package has been modified to use constrained types. The result is that the `Name` function of the `Test_Case` object no longer returns an object of type `Ada.Strings.Unbounded.String_Access`. It now returns an object of type **`AUnit`**`.Test_Results.Test_String`.

   Here is an example of how to change its implementation:

```
function Name (T : List_Test_Case) return Ada.Strings.Unbounded.String_Access
is
begin
   return new String'("My test case");
end Name;
```

   Should be changed to:

```
function Name (T : My_Test_Case) return AUnit.Test_String is
begin
   return AUnit.Test_Results.Format ("My test case");
end Name;
```

4. Automation of the conversion

   Note that some of the changes described above can be automated by scripts.

   For example, changing the with clauses and modifying the Name function can be performed by the following sed script, that can be adapted to the specific structure of the existing tests.

```
#!/usr/bin/sed
: first_with_clause
s/with \(\([,][ ]*\)\?AUnit[^,;]*\)*/with AUnit/
t following_with_clauses
n
```

```
b first_with_clause

: following_with_clauses
n
s/with \(\([,][ ]*\)\?AUnit[^,;]*\)*; *//
t following_with_clauses
s/function Name/&/
t function_Name_match
b following_with_clauses

: function_Name_match
s/return .*String_Access/return AUnit.Test_String/
t spec_matched
n
b function_Name_match

: spec_matched
s/;/&/
T body_matched
b end

: body_matched
s/ *end/&/
t end
s/new String[']/Aunit.Test_Results.Format /
t end
n
b body_matched

:end
```

The script can be used by the following command to modify all files in dir ‘`src`’. It will place the results into ‘ `../AUnit2_tests_repository/src`’:

```
for f in ‘src/*.ad[bs]‘; do
  sed -f from_aunit1_to_aunit2.sed $f > ../AUnit2_tests_repository/$f;
done
```

11 Using AUnit with Restricted Run-Time Libraries

AUnit 2 is a reimplementation of the original AUnit so that it can be used in environments with restricted Ada run-time libraries, such as ZFP and the cert run-time profile on Wind River Systems PSC ARINC-653. The patterns given in this document for writing tests, suites and harnesses are not the only patterns that can be used with AUnit, but they are compatible with the restricted run-time libraries provided with GNAT Pro.

In general, dynamic allocation and deallocation must be used carefully in test code. For the cert profile on PSC ARINC-653, all dynamic allocation must be done prior to setting the application partition into "normal" mode. Deallocation is prohibited in this profile. For the default ZFP profile, dynamic memory management is not implemented, and should not be used unless you have provided implementations as described in the GNAT Pro High Integrity User Guide.

Additional restrictions relevant to the default ZFP profile include:

1. Normally AUnit will list any unexpected exceptions that occur during test execution. However, since the default ZFP profile does not support exception propagation, control is instead passed to the user last chance handler. As in all ZFP profiles, such a last chance handler is required.
2. AUnit requires `GNAT.IO` provided in '`g-io.ad?`' in the full or cert profile run-time library sources (or as implemented by the user). Since this is a run-time library unit it must be compiled with the gnatmake "-a" switch.
3. The AUnit framework has been modified so that no call the the secondary stack is performed. However, if the unit under test, or the tests themselves require use of the secondary stack, then the test suite must export symbol `__gnat_get_secondary_stack`. This is not actually used unless the application or unit tests require the secondary stack, in which case it must be fully implemented.
4. Failed assertions do not abandon execution of the calling test routine in ZFP profiles that do not support exception propagation. A functional form of the `Assert` subprogram allows the calling routine to determine whether to continue or abandon its further execution. This behavior is selected during compilation when the *makefile*'s `SUPPORT_EXCEPTION` variable is set to `no`.
5. The timed parameter of the Harness `Run` routine has no effect when used with the ZFP profile. This behavior is selected during compilation when the *makefile*'s `SUPPORT_CALENDAR` variable is set to `no`.

12 Installation and Use

AUnit 2 now contains support for limited run-times such as zero-foot-print (zfp) and certified run-time (cert). However, some functionalities have to be disabled on those run-times: the cert run-time does not provide Ada.Calendar and zfp does not support exceptions, and does not provide Ada.Calendar either.

The AUnit library can be installed to take into account those limitations.

12.1 Installing AUnit on UNIX systems

- Extract the archive:

```
$ gunzip -dc aunit-2.01-src.tgz | tar xf -
```

- To build and install AUnit for a full Ada run-time:

```
$ cd aunit-2.01-src
$ make INSTALL=<gnat-root> install
```

  Where <gnat-root> is for example /opt/gnat/6.0.1

- Specific installation:

  The AUnit makefile supports some specific options, activated using environment variables. The following options are defined:

  - INSTALL: defines the AUnit base installation directory, should always be set.
  - TOOL_PREFIX: defines the gnat tools prefix to use. For example, to compile AUnit for powerpc VxWorks, TOOLS_PREFIX should be set to powerpc-wrs-vxworks. If not set, the native compiler will be used.
  - RTS: defines the run-time used to compile AUnit. When set to zfp, this automatically defines SUPPORT_CALENDAR and SUPPORT_EXCEPTION (see below).
  - SUPPORT_CALENDAR: takes the values yes (default) or no. If the compiler or the run-time used do not provide Ada.Calendar, then you should set this variable to 'no'.
  - SUPPORT_EXCEPTION: takes the values yes (default) or no. If the compiler or the run-time used do not provide exceptions support, then you should set this variable to 'no'.

  Example of installation of AUnit to /opt/gnatpro/5.04a1, for a cross ppc vxworksae compiler using the zfp run-time,

```
$ cd <build-dir>
$ make INSTALL=/opt/gnatpro/5.04a1 RTS=zfp \
    TOOL_PREFIX=powerpc-wrs-vxworksae install
```

- To test AUnit:

  The AUnit test suite is in the test subdirectory of the source package. In order to build and run the AUnit test suite, first install the AUnit library, then:

```
$ cd test
$ make
```

  The test suite's makefile supports the following variables: * RTS * TOOL_PREFIX

12.2 Installing AUnit on Windows systems

On Windows, an install-shield wizard is available to easily install AUnit. This install shield will ask some questions during the installation:

- Selection of the compiler: the install-shield will try to detect all GNAT compilers available in your system. You can directly select one of the detected compilers or enter the path of the desired compiler. The entered path is the root path of the compiler: <path>/bin/*gnatmake.exe should be present.

- Selection of the installation directory: by default, AUnit is installed in the same root directory as the selected compiler. Enter another directory at this stage if you want to install AUnit in another directory.
- Selection of the run-time: if you want to compile AUnit with a specific run-time, enter the run-time at this stage.
- Support for exceptions: select the support for exceptions in AUnit. If the selected run-time does not support exceptions, you should disable the exception support in AUnit at this stage.
- Support for Ada.Calendar: select the support for Ada.Calendar in AUnit. If the selected run-time does not provide Ada.Calendar, you should disable the Ada.Calendar support in AUnit at this stage.

After entering the information above, the install-shield will build and install aunit in the selected installation directory.

12.3 Installed files

The AUnit library is installed in the specified directory (<gnat-root> identifies the root installation directory as specified during the installation procedures above):

- the aunit.gpr project is installed in <gnat-root>/lib/gnat
- the AUnit source files are installed in <gnat-root>/include/aunit
- the AUnit library files are installed in <gnat-root>/lib/aunit
- the AUnit documentation is installed in <gnat-root>/share/doc/aunit
- the AUnit examples are installed in <gnat-root>/share/examples/aunit

13 GPS Support

The GPS IDE has a menu **Edit -> Unit Testing** to generate template code for test cases, test suites and harnesses. The current templates are for AUnit 1.x, so will need to be migrated as described in “Migrating Tests from AUnit 1 to AUnit 2”.