Adapting ACATS to the Ahven Testing Framework

Dan Eilers[1] and Tero Koskinen[2]

[1] Irvine Compiler Corp.
http://www.irvine.com
dan@irvine.com
[2] tero.koskinen@iki.fi

Abstract. The Ada Conformity Assessment Test Suite (ACATS) includes thousands of individual executable test programs, but no test driver or tools for grading the output. We show how ACATS can be adapted to work with the Ahven testing framework, resulting in a single easy-to-build executable program that combines the executable ACATS tests, runs them in order, and grades and summarizes the test results. Our goal is a highly portable and automated ACATS driver, and as a side benefit we obtain a somewhat more stressful test capability for Ada compilation systems.

Keywords: ACATS, Ahven, testing framework, test harness, test automation, conformance, conformance testing, Ada, compiler.

1 Introduction

The Ada Conformity Assessment Test Suite (ACATS) [1] is a publicly available test suite intended to check Ada compilers for conformance with the Ada [2] standard. It is derived from the original Ada Compiler Validation Capability (ACVC) sponsored by the former Ada Joint Program Office (AJPO).

Although ACATS is intended to be relatively straightforward to use, it requires much more effort than simply typing a few commands and waiting for the results. This is partly because there is no supplied test driver nor any tools for grading the output. ACATS users are expected to develop their own driver scripts and analysis tools, which presents a high barrier to use, especially for those who are accustomed to downloading and building large software packages with a single command. We look to automated testing frameworks as a solution.

Testing frameworks for Ada have become popular relatively recently, notably including Ahven [3,4] developed by Tero Koskinen (co-author of this paper) and AUnit [6] developed by Ed Falis. Both frameworks were inspired by the JUnit testing framework for Java, and both are open source Ada packages in the Debian GNU/Linux distribution [5,7]. These frameworks work by combining a collection of individual test procedures into a single executable program, where they are

A. Romanovsky and T. Vardanega (Eds.): Ada-Europe 2011, LNCS 6652, pp. 75–88, 2011.

run in order and graded, with a summary of the results produced at the end. A single "Ada make" command suffices to build the combined test program, with no need for unportable shell scripts. We focus on the Ahven framework because it avoids the need to write a test wrapper for each individual ACATS test.

Combining ACATS test programs is something that the ACATS User's Guide envisions for efficiency reasons. However, our concern is not efficiency. Instead we are concerned with automating the process in a portable manner. And as a side benefit, combining the tests creates the possibility of exposing runtime errors that might otherwise have gone undetected.

ACATS includes both "positive" and "negative" tests. That is, tests that are intended to compile and run successfully, and those with errors that are intended to be rejected by the compiler. We have found that the positive tests are well suited for automation using Ahven. The negative tests, however, are not. Negative tests pose grading difficulties because they typically have multiple intentional errors per test. The grader must verify that the compiler has detected each of those, without rejecting any legal constructs. Negative tests are also not suitable for testing frameworks such as Ahven that expect to successfully compile and run each test case. So this paper is concerned only with the positive tests.

Fortunately, the positive ACATS tests are both the easiest to grade as well as the most interesting, for users who are especially concerned with correctness of compiled Ada code, since negative tests do not exercise the dynamic semantics, optimization, and code generation phases of the compiler. Users who may have modified the compiler's run-time system, or who are using an unusual set of compiler switches would also benefit most from the positive tests. Some previous language test suites, such as those for Cobol and Fortran did not even include any negative tests [12]. There are however occasional cases where negative tests can have value with regard to preventing incorrectly compiled code, such as where a negative test enforces a language rule designed to prohibit a dangerous combination of features.

We envision Ahven being used with ACATS primarily for informal compiler testing, particularly by those who have never used ACATS before because they considered it too much trouble. So we are not bothered by a few unusual tests that may require omission or special handling. We believe that the minor ACATS test modifications we describe would be beneficial to incorporate into a future version of the test suite, and are happy to make them available upon request.

Section 2 describes ACATS and existing scripting capabilities; Section 3 describes Ahven and how it can be integrated with ACATS; Section 4 describes some elaboration issues requiring test modification; Section 5 describes how using a framework can make ACATS more effective; Section 6 compares Ahven with AUnit; and Section 7 gives the conclusions.

2 Background and Related Work

> I've written the following simple routine, which may separate the "man-compilers" from the "boy-compilers". *Donald Knuth*

Unfortunately, Ada compiler conformance testing is not as simple as Knuth's clever 10-line test program for Algol 60 compilers [8]. But perhaps his novel idea of a single executable test program is not as far-fetched as it might seem.

For Ada, we have the ACATS test suite, which was designed from the beginning to include thousands of individual test programs, intending to provide a comprehensive conformance assessment [9]. Composing a test suite from many tests has obvious advantages. It produces meaningful results for partially conforming compilers, and it simplifies analysis of failing tests. But such a large collection of tests requires some sort of test harness and automated grading tools in order to be cost effective.

ACATS does not come with a modern testing framework such as Ahven or AUnit. This comes as no surprise, since ACATS and the ACVC before it predate such frameworks. The ACATS User's Guide [10] was written with the expectation that users of the suite would use custom-developed shell scripts to compile, run, and grade the positive tests. Ideally, such scripts would be provided with ACATS. But it is difficult if not impossible to create portable shell scripts that would work with any potential compiler on any potential operating system.

A significant milestone in ACATS automation was a shell script developed by Laurent Guerby for running and grading the ACATS positive tests using the gcc Ada compiler. This script was incorporated into the testsuite of gcc 3.4 and later [11, Section 2.3.4]. It has provided various individuals and organizations who rehost and/or retarget the gcc Ada compiler with an effective means of demonstrating that the compiler and run-time system are working properly. It also provides a fully automated compiler test capability for system administrators, compiler testers, and end users who download and build the gcc Ada compiler from source code. Most such users would not be using ACATS if it were not fully automated.

The shell scripts used by the gcc testsuite rely on textual analysis of the ACATS output, using the "Expect" tool. Although the shell interpreter and the Expect tool are both widely available, our goal is to have a more portable pure Ada solution that eliminates any textual analysis by hooking directly into the ACATS grading mechanism. We also wish to have a compiler-independent solution.

2.1 Nature of the ACATS Tests

The ACATS positive tests are supplied in a directory structure organized by chapter of the Ada Reference Manual. Some tests have multiple compilation units in the same file, and some tests have multiple compilation units spread across several files. Each test, and each compilation unit within a test, generally has a unique name, so all such tests could conceivably be compiled into the same Ada environment without interference.

The ACATS User's Guide specifies that the tests should be compiled and run in the order given, but generally the tests are independent of each other, and the order of compilation and execution doesn't matter. So a compilation

system's "make" facility for automatically determining compilation order would normally be effective. In fact, the gcc testsuite mentioned earlier uses gnatmake to determine compilation order for tests with multiple files.

There are a few tests for RM Chapter 10 that include multiple compilation units with the same name, for the purpose of testing that a later compiled unit properly replaces a previously compiled unit with the same name. These tests can be handled by submitting their source files to the compilation system in alphabetical order, except for test *ca14028e* which may require special handling.

Most ACATS tests are self-contained. However, there are approximately fifty support packages, referred to as foundation code, that are used by multiple tests. Sharing code between bundled tests presents an issue of package elaboration as discussed below.

There are a few *Text_IO* tests in RM Annex A that interfere with each other, such as when one test calls *Set_Input* to change the default input file, without restoring it, causing a later test to fail. The affected tests are *ce3806a*, *ce3806b*, *ce3706d*, *ce3605c*, *ce3405c*, *ce3202a*, *ce3411c*, *ce3413c*. We have not yet identified the best solution for these tests.

The ACATS tests generally don't use command-line arguments, which would prevent them from being bundled. One exception is test *cxaf001* which may require special handling.

2.2 Bundling Test Programs

The ACATS User's Guide was written with the expectation that the tests will normally be compiled and run individually. But Section 5.5.3 specifically describes the possibility of bundling multiple tests into a single executable for efficiency reasons.

> 5.5.3 Bundling Test Programs
> In some situations, the usual test processing sequence may require an unacceptable amount of time. For example, running tests on an embedded target may impose significant overhead time to download individual tests. In these cases, executable tests may be bundled into aggregates of multiple tests. A set of bundled tests will have a driver that calls each test in turn; ACATS tests will then be called procedures rather than main procedures. No source changes in the tests are allowed when bundling; that is, the only allowed change is the method of calling the test.
> All bundles must be approved by the ACAL (and, if necessary, the ACAA) to qualify for a conformity assessment. It is the responsibility of the user to identify the tests to be bundled and to write a driver for them.

Bundling of test programs is facilitated in Ada by the lack of any syntactic distinction between an Ada main program and an Ada subprogram. In Pascal or

Fortran, for example, the main program uses the **program** keyword to distinguish it from ordinary subprograms. Similarly, in C and C++, the main program is distinguished by its name, *main*. Java is more like Ada, in that any Java class can be treated as the main program.

So bundling of existing Ada test programs requires no changes to the tests. Given three test programs, *P1*,*P2*,*P3*, we can simply do:

```
with P1, P2, P3;
procedure Main_Driver is
begin
   P1;
   P2;
   P3;
end;
```

An obvious drawback of this solution is that if an unexpected exception is propagated out of one of the earlier tests, the remaining tests will not be run. This can be solved by adding a begin-end block with an exception handler around each call. But that starts to get messy. If we want to add protection from infinitely looping tests, it gets even messier. Keeping track of pass/fail statistics makes it messier still. We would probably want to do some refactoring, to isolate all the per-test support code in one location, which is exactly what a testing framework is all about.

Caution Regarding Undistinguished Main Programs. Although Ada's undistinguished main programs are useful for test bundling, they can surprise the unwary in other situations. An Ada main program can call itself recursively, either directly or indirectly, just like any other Ada subprogram. So the storage for the main's local variables will likely be stack based rather than being statically allocated. This raises the possibility of stack overflow when the main program includes large array variables on systems with fixed stack limits. For similar reasons, a subprogram nested in the main program may use static links to reference the main's local variables, rather than global references, affecting the nested subprogram's low-level calling signature.

An additional surprise is that in order to prevent dangling pointer references, Ada has the notion of accessibility levels, where library subprograms are considered to be one level deeper than library packages. Ada 95 had a rule (later repealed in Ada 2005) preventing type extension of tagged types at a deeper accessibility level than the parent type (RM95 3.9.1(3)). This rule disallowed extending a library-level tagged type inside the main subprogram, just like any other library subprogram [13, Section 15.1].

2.3 Hooks for Attaching ACATS Tests to a Testing Framework

ACATS tests generally have a standard structure. Section 4.6 of the ACATS Users Guide shows:

Executable tests (class A, C, D, and E) generally use the following format:

```
with Report;
procedure Testname is
   <declarations>
begin
   Report.Test ("Testname", "Description ...");
   ...
   <test situation yielding result>
   if Post_Condition /= Correct_Value then
      Report.Failed ("Reason");
   end if;
   ...
   Report.Result;
end Testname;
```

There is a package *Report*, which is common to all tests. At the beginning of each test, there is a call to *Report.Test*. In the body of each test there are one or more possible calls to *Report.Failed*, whenever an error is detected. At the end of each test there is a call to *Report.Result*. A test is considered to pass if there were no calls to *Report.Failed* before the call to *Report.Result*.

This *Report* package is convenient for our purposes, since it provides the necessary hooks to integrate a testing framework simply by making minor modifications to the *Report* package body, without needing to make any modifications to the tests, in general. The exceptional cases are noted below.

3 Integration with Ahven

> Where the primary concern in OO is encapsulation, the primary concern in data-driven programming is writing as little fixed code as possible [14].
> *Eric S. Raymond*

We have three main goals for integration of ACATS with a testing framework. The first is to avoid modifications to the tests, except in a few cases where necessary. This is easily accomplished by adding a call to *Ahven.Fail* in the ACATS *Report* package body, in procedure *Result*, when a test is determined to have failed.

ACATS actually categorizes test results into four states, passed, failed, not-applicable, and tentatively-passed. This presents an issue of how the not-applicable and tentatively-passed states should be mapped to Ahven pass-fail states. Normally, these are considered as passing. Alternatively, their results could be highlighted for hand analysis by treating them as failing, using the message parameter to *Ahven.Fail* to record the details.

The second goal is that the test driver should be hierarchical, reflecting the directory structure of ACATS, where tests are grouped by RM chapter. Ahven

supports arbitrarily deep test-suite hierarchies, although only two levels are used for ACATS.

The third goal is to keep the test driver as simple as possible. Ideally, the test driver will be data driven, using a framework-independent Ada aggregate, with one line in the aggregate per test. In particular, we want to avoid having to perform multiple operations in the driver for each test. We also want to avoid having to write a wrapper subprogram for each test.

3.1 Framework-Independent Representation of Tests

Suppose for example we have six tests, *P1*,*P2*,*P3*,*Q1*,*Q2*,*Q3*, partitioned into two groups, *P* and *Q*, of three tests each.

We would like to define a type *One_Test*, which is a record containing the necessary information about each test, consisting of an access to the test procedure, and an access to the test name. Then we can declare type *Suite_Type*, which is an array of *One_Test*. For example:

```
package Test_Suite is

  type Proc_Access is access procedure;
  type One_Test is record
     Proc: Proc_Access;
     Name: access String;
  end record;
  type Suite_Type is array(Natural range <>) of One_Test;

  generic
     Suite_Name: String;
     Suite_Array: Suite_Type;
  package Suites is
  end Suites;

  function "+"(S: String) return access String;
end Test_Suite;

package body Test_Suite is

  function "+"(S: String) return access String is
  begin
     return new String'(S);
  end;
end Test_Suite;
```

This example includes a generic package *Suites*, which can be instantiated with the name of a test suite, and an array aggregate providing the data for each test in the suite. So we would define *P_Suite* as:

```
with P1, P2, P3;
with Test_Suite; use Test_Suite;
package P_Suite is new Test_Suite.Suites(
     Suite_Name => "P",
     Suite_Array => (
        (P1'access, +"P1"),
        (P2'access, +"P2"),
        (P3'access, +"P3")));
```

And *Q_Suite* would similarly be defined as:

```
with Q1, Q2, Q3;
with Test_Suite; use Test_Suite;
package Q_Suite is new Test_Suite.Suites(
     Suite_Name => "Q",
     Suite_Array => (
        (Q1'access, +"Q1"),
        (Q2'access, +"Q2"),
        (Q3'access, +"Q3")));
```

This example meets our goal of defining each test suite with an aggregate containing one line per test.

3.2 Connecting with Ahven

Now that we have a framework-independent way of representing our set of tests, we can write some glue code to connect our *Test_Suite* package to Ahven.

In Ahven, each sub-suite of individual tests is created by deriving a new controlled type from *Test_Case*, and overriding its *Initialize* routine. Inside *Initialize*, the sub-suite's name is specified by calling *Set_Name*. Then each test in the sub-suite is registered by calling *Add_Test_Routine*, passing an access to the test, and the name of the test. Outside of *Initialize*, a framework is created by calling *Create_Suite*, and each sub-suite is added to the framework by calling *Add_Test*. The framework, which now includes all of the tests in all of the sub-suites, is run, graded, and summarized by calling *Run*.

The glue code is as follows:

```
with Ahven.Framework;
with Test_Suite;
generic
   S : Ahven.Framework.Test_Suite_Access;
   with package The_Suite is new Test_Suite.Suites(<>);
package One_Suite is
   type Test is new Ahven.Framework.Test_Case with null record;
   procedure Initialize (T: in out Test);
end One_Suite;
```

```
package body One_Suite is

   procedure Initialize (T: in out Test) is
   begin
      T.Set_Name(The_Suite.Suite_name);
      for I in The_Suite.Suite_Array'range loop
        T.Add_Test_Routine(
          Ahven.Framework.Simple_Test_Routine_Access(
                The_Suite.Suite_Array(I).Proc),
          The_Suite.Suite_Array(I).Name.all);
      end loop;
   end;

begin
   Ahven.Framework.Add_Test (S.all, new Test);
end One_Suite;

with P_Suite;
with Q_Suite;
with One_Suite;
with Ahven.Framework;
package All_Suites is
   S : Ahven.Framework.Test_Suite_Access :=
     Ahven.Framework.Create_Suite ("all tests");

   package Suite_P is new One_Suite(S, P_Suite);
   package Suite_Q is new One_Suite(S, Q_Suite);
end All_Suites;

with All_Suites; use All_Suites;
with Ahven.Text_Runner;
with Ahven.Framework;
procedure Main_Driver is
begin
   Ahven.Text_Runner.Run (S);
   Ahven.Framework.Release_Suite (S);
end Main_Driver;
```

The result of running *Main_Driver* will be the output from each of the individual tests, followed by an Ahven summary report such as the one below. The right-hand column shows the execution time taken for each test.

```
Passed :  6
 all tests:
  P:
   P1                                          PASS      0.00000s
```

```
   P2                                          PASS      0.00000s
   P3                                          PASS      0.00000s
  Q:
   Q1                                          PASS      0.00000s
   Q2                                          PASS      0.00000s
   Q3                                          PASS      0.00018s
```

Now we can simply replace the *P_Suite* and *Q_Suite* shown above in package *All_Suites* with one suite for each subdirectory in ACATS, which is organized by chapter in the Ada RM. This achieves our goal of a single program that will run, grade, and summarize all executable ACATS tests. Since this program includes a with-clause for each individual ACATS test, it is easy to build, with just a single "Ada make" command.

4 Elaboration Issues in Shared ACATS Support Code

There are however some technical details to resolve before we can declare success. As noted above, some ACATS tests depend on shared foundation packages. When such packages use package elaboration to initialize their variables, we must ensure that the shared package is properly re-initialized before running the next test that depends on it.

There are three such foundation packages, along with the tests that depend on them, that need modification.

Specifically, foundation package *f390a00* includes the declaration

```
Display_Count_For : Display_Counters := (others => 0);
```

This package is shared between three tests, *c390a011*, *c390a022*, and *c390a031*.

Foundation package *f393a00* includes the declaration

```
Finger : Natural := 0;
```

This package is shared between four tests, *c393a02*, *c393a03*, *c393a05*, and *c393a06*.

Foundation package *fb40a00* includes the declarations

```
AlphaNumeric_Count,
Non_AlphaNumeric_Count : Natural := 0;
```

This package is shared between four tests, *cb40a01*, *cb40a021*, *cb40a031*, and *cb40a04*.

In each case there is no existing means for reinitializing the variables, so we propose to add a new initialization procedure to each of these three foundation packages, and add a call to these new procedures at the beginning of each of these eleven dependent test cases.

4.1 Shared Support Package TCTouch

There is a special shared support package, *TCTouch*, which is used by more than 40 tests. It verifies that each test performs particular actions in a particular order. It includes the declaration

```
Finger : Natural := 0;
```

Happily, there does exist an initialization procedure *Flush* that resets variable *Finger* to zero. Unfortunately, most tests that use package *TCTouch* do not call Flush. Instead they simply rely on the default initialization. We must ensure that *Flush* is called at the start of every test that uses package *TCTouch*. This could be done with a small change to 40+ tests. Alternatively we could ensure that *Flush* is called before every test, whether it needs to be or not, by inserting one call in procedure *Test* of the ACATS *Report* package.

With *Flush* called at the start of every test, we must also ensure that there are no cases where calls are made to procedure *Touch* in package *TCTouch* during initialization of the test's local variables. This affects two tests, *c393001* and *c761013*.

Test *c393001* includes the declarations

```
Short : C393001_1.Breaker'Class -- Basic_Breaker
        := C393001_2.Construct( C393001_2.V440, C393001_2.A5 );
Sharp : C393001_1.Breaker'Class -- Ground_Fault
        := C393001_3.Construct( C393001_2.V110, C393001_2.A1 );
Shock : C393001_1.Breaker'Class -- Special_Breaker
        := C393001_4.Construct( C393001_2.V12,  C393001_2.A100 );
```

These calls to *Construct* during initialization result in calls to *TCTouch.Touch*. So we need to move these three declarations into a newly created declare block in this test's main procedure, just after the call to *Report.Test*, to delay their effect.

Test *c761013* includes the declaration

```
Outer : Ctrl;
```

where *Ctrl* is a controlled type with an initialization procedure that calls *TCTouch.Touch*. So we need to move this declaration, and also procedure *Subtest_3*, which references it, to a newly created declare block in this test's main procedure, just after the call to *Report.Test*, to delay its initialization.

A more subtle issue is that we must ensure that no calls are made to *TCTouch.Touch* during the library package elaboration of any test. This is because test bundling causes the elaboration of all library packages to be done before any individual test is started. It would defeat our attempt to reinitialize package *TCTouch* before every test if calls were being made to *Touch* from the combined elaboration code of multiple tests.

This issue affects one test, *c3a2001*, which includes the library package elaboration code:

```
Short : C393001_1.Breaker'Class -- Basic_Breaker
        := C393001_2.Construct( C393001_2.V440, C393001_2.A5 );
Sharp : C393001_1.Breaker'Class -- Ground_Fault
        := C393001_3.Construct( C393001_2.V110, C393001_2.A1 );
Shock : C393001_1.Breaker'Class -- Special_Breaker
        := C393001_4.Construct( C393001_2.V12,  C393001_2.A100 );
```

These calls to *Construct* during elaboration result in calls to *TCTouch.Touch*. So we need to move them into a newly created initialization procedure in package *C3A2001_5*, which we call just after the call to *Report.Test*, to delay their effect.

4.2 Tests with Unusual Organization

Test *cd5003a* is organized somewhat differently than usual. It calls *Report.Test* and *Report.Failed* in the initialization code of a library package, rather than in the test's main procedure. It is not difficult to reorganize this test into the usual format, without affecting its objective, by replacing the package elaboration code with a newly created procedure called from the main procedure. Such reorganization is necessary in order to avoid calling the *Report* package during any test's elaboration, since test bundling has the effect of combining elaboration code for all tests.

There are six tests, *c761001*, *c761010*, *c94004a*, *c94004b*, *c94004c*, and *c94005a* that call *Report.Result* as part the library unit finalization. There are four tests, *c39006f3m*, *ca5004a*, *ca5006a*, and *ca5004b2m* that call *Report.Test* in library package initialization. These tests may require special handling.

5 Bundling As a Compilation System Stressor

Bundling of tests offers a nice side benefit of increasing the stress on a compilation system, thereby maximizing the benefit of the test suite to find anomalies in the compilation system. In particular, bundling has the potential to uncover cases where the state of the run-time system at the completion of one test isn't pristine enough to run the next test. Bundling all ACATS positive tests into a single executable program also has the potential to uncover capacity issues or capacity-related performance issues in the various tools, such as the library system, compilation-order tool, linker, debugger, etc.

5.1 Repeatability and Ordering of Tests

Using a testing framework makes it trivial to stress the compilation system even further, by executing tests multiple times within the same program, to ensure repeatability. Most ACATS tests can be restarted, but a preliminary investigation suggests that some tests may require minor modifications.

Frameworks also have the potential to easily alter the order in which the tests are run. Doing so may expose subtle compiler or run-time system anomalies that might not otherwise show up.

5.2 Making Positive Tests Out of Negative Tests

We have ignored the negative tests, but Robert Eachus has suggested [15] that by removing the errors from the negative tests, we would be left with useful tests which could be treated as additional positive tests. Preliminary investigation shows this to be feasible and worthwhile, although substantial manual effort is required.

6 Ahven vs. AUnit

AUnit is quite similar to Ahven, but the way tests are registered is slightly different. Ahven accepts a parameterless test procedure, where AUnit does not. In procedure *Initialize* above, we have:

```
Ahven.Framework.Simple_Test_Routine_Access(
      The_Suite.Suite_Array(I).Proc),
```

where *The_Suite.Suite_Arr(I).Proc* designates the actual parameterless ACATS test procedure. AUnit registration requires writing a wrapper procedure for each test, which includes a parameter of type *AUnit.Test_Cases.Test_Case'Class*.

Ada does not seem to have a way of automating creation of these wrapper procedures that AUnit requires, given our array aggregate of access to parameterless procedures. Ada does allow procedures to be created dynamically using generics, but such procedures would be at the wrong accessibility level. Ada also allows procedures to be created dynamically using allocators to protected records containing a protected procedure. But AUnit does not allow registering protected procedures.

7 Conclusions

We have shown the ability to integrate the Ahven testing framework with the ACATS test suite, providing a portable test driver. This is done with only minor modifications to the ACATS test suite. The test driver is data driven, with one line of test description per test using a framework-independent Ada aggregate, organized by RM chapter. The resulting single program is easy to build using "Ada make", and it automatically runs the tests and grades and summarizes the results. Combining tests has the side benefit of providing a more stressful test of the compilation system. Improving the cost/benefit ratio of running ACATS may increase the usage of this valuable resource.

Acknowledgments. The authors would like to thank the anonymous referees for their helpful comments.

References

1. Brukardt, R.L.: Ada Conformity Assessment Test Suite (ACATS), http://www.ada-auth.org/acats.html
2. Taft, S.T., Duff, R.A., Brukardt, R.L., Plödereder, E., Leroy, P.: Ada 2005 Reference Manual. LNCS, vol. 4348. Springer, Heidelberg (2006)
3. Koskinen, T.: Ahven developer, http://sourceforge.net/projects/ahven
4. Koskinen, T.: Ahven 1.8 announcement on comp.lang.ada newsgroup reprinted in Ada User Journal. Ada Europe 31(3), 159–161 (2010)
5. Buerki, R., Rueegsegger, A.-K.: Ahven package maintainers in Debian GNU/Linux, http://packages.debian.org/ahven
6. Falis, E.: AUnit developer, http://libre.adacore.com/libre/tools/aunit
7. Leake, S.: LibAunit package maintainer in Debian GNU/Linux, http://packages.debian.org/libaunit
8. Knuth, D.E.: "Man or boy?". ALGOL Bulletin 17, 7 (1964); 19, 8–9 (January 1965), Reprinted as ch. 6 of Selected Papers on Computer Languages. Center for the Study of Language and Information, Stanford, California (2003)
9. Goodenough, J.B.: The Ada Compiler Validation Capability. Computer 13(6), 57–64 (1981), doi:10.1109/C-M.1981.220496
10. Brukardt, R.L.: Ada Conformity Assessment Test Suite (ACATS) User's Guide, Version 3.0 (2008), http://www.ada-auth.org/acats-files/3.0/docs/ACATS-UG.PDF
11. Brenta, L., Leake, S.: Debian Ada Policy. 5th edn (May 29, 2010), http://people.debian.org/~lbrenta/debian-ada-policy.html
12. Oliver, P.: Experiences in Building and Using Compiler Validation Systems. In: Proc. of AFIPS Conf., NCC, vol. 48, pp. 1051–1057 (1979)
13. English, J.: Ada 95: The Craft of Object-Oriented Programming (2001), http://www.it.bton.ac.uk/staff/je/adacraft
14. Raymond, E.S.: The Art of Unix Programming. Pearson Education, Inc., London (2004)
15. Eachus, R.: Personal communication (May 2010)