Measuring the Effectiveness of ACATS

Geoff T. Smith
IBM Rational Software
15300 SW Koll Parkway
Beaverton, OR 97006
gsmith@us.ibm.com

ABSTRACT

This paper reviews the effectiveness of the Ada Conformity Assessment Test Suite (ACATS) for testing the implicit components of an Ada application. Rough measures are achieved on the basis of both requirements coverage and structural coverage.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Validation
D.3.0 [**Programming Languages:General**]: Standards
D.2.5 [**Testing and Debugging**]: Testing Tools (coverage testing)

General Terms: Measurement, Verification.

Keywords: ACATS, runtime, conformance, validation, conformity assessment

1. INTRODUCTION

Technically savvy users generally have two reactions to a test suite. The first one is relief that such a suite exists. And later, after a little more experience, comes the inevitable question: "How good is that suite, anyway?"

Ada is a unique programming language in having an industry-accepted, international standard for conformity. [1], [2] The standard specifies the Ada Conformity Assessment Test Suite (ACATS), and the procedures and protocols for conducting the tests and obtaining a certificate of conformity. The test suite and user's manual are downloadable from the web [3].

Although passing the ACATS suite is necessary for an Ada compilation system, the ACATS as a test suite is not exempt from the question of how "good" it is. This paper discusses an investigation into how effective the ACATS is from the viewpoint of a programming team using an Ada compiler.

2. COVERAGE MEASURES

The ***effectiveness*** of a functional test suite can be measured by the coverage it provides. Given an application, a suite is more effective as it tests more of that application. Test suites tend to become more effective over time as new tests are added – generally to cover bits that were overlooked in previous versions of the suite.

There are two major classifications of coverage.

- Requirements coverage
- Structural coverage

Requirements coverage measures how much of the product requirements are covered by tests in the suite. Requirements coverage is simple in concept – simply list each product requirement, and determine which (if any) tests in the suite test that requirement. The resulting count of covered and uncovered requirements can be converted to a percentage.

In practice, analysis of tests vs. requirements becomes complicated. Too often, product requirements are vague, too high-level, untestable, or simply incomprehensible. The utility of requirements coverage analysis hinges on the quality of the requirements documentation, and depends heavily on having a disciplined process for maintaining product requirements at a sufficient level of detail. With today's preoccupation with rapid development, this kind of process is increasingly limited to mission- and safety-critical projects.

Structural coverage measures how much of the executable code is tested on some physical measure. This measure can be taken on a source-code basis (for example, how many lines of source of the application are actually executed), or on an object-code basis (i.e., number of machine instructions).

Further discussion of structural coverage techniques (sometimes called execution coverage), relative merits, interpretation, and pitfalls is beyond the scope of this paper. This is a well-developed topic, with numerous papers available both in industry publications and the web. Many software tool vendors have excellent discussions of coverage techniques on their web sites.

3. ANATOMY OF AN ADA EXECUTABLE

At first glance, an Ada executable is made up of many lines of source code that are compiled into machine instructions, which are then combined somehow to produce an executable.

Of course, this is not the whole story. A little dissection is in order here. Looking closely at an executable reveals that it consists of numerous machine code instructions, but not all these instructions come from the programming team's source code. An executable almost always includes machine instructions that are implicitly included by the compilation system.

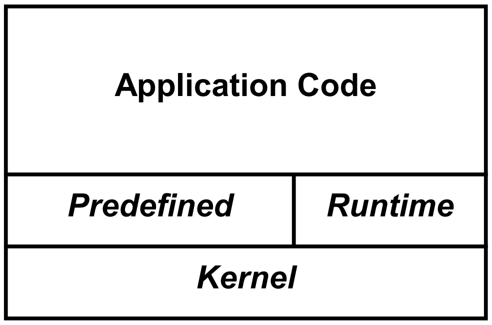


Figure 1. Explicit and Implicit Code.

3.1 Application Code

This is the source code written by a programming team and explicitly included in the executable.

3.2 Predefined Code

Predefined units are those provided with the compiler, but explicitly included in the Ada closure. One example is the package `Ada.Text_Io`. Predefined units include the language-defined library packages explicitly specified in the Ada Language Reference Manual (LRM) [4], typically along with other, vendor-defined packages.

3.3 Runtime

The runtime includes compiler "assist" functions (e.g., the implementation of `Integer'Image`) and the implementation of Ada constructs such as tasking and exception handling. These are functions that are invoked outside the Ada "with" closure of the executable.

3.4 Kernel

The kernel represents the underlying OS (and device drivers) when the Ada compilation system targets a system with coexisting executables (such as Unix or Linux). This function is generally not written in Ada except in "bare-board" embedded systems.

4. IMPLICATIONS FOR TESTING

When it comes to testing, the programming team needs to test their own application code (or hire someone else to conduct the testing). But beyond this, they must also consider how to treat the implicit portions of the executable.

Most projects simply rely on the vendor's testing or the incidental testing of the runtime that occurs in their own tests.

This is generally not a high-risk proposition, especially for a compilation system supplied by a major vendor such as Rational. The vendor-supplied parts of the runtime benefit from the use and experience gained by the vendor's other customers, and also from good coding practices forced by the vendor's necessity to support multiple architectures. In addition, the vendor may (like Rational) maintain its own sizeable test suites to supplement ACATS testing.

On the other hand, some projects are required to document or even conduct testing of the entire application, explicit and implicit. This is more typical of safety-critical systems.

In either case – whether relying on vendor testing, or running tests in-house – the ACATS is the logical starting point. Conceptually, the standard Ada tests should cover the implicit parts of an application for the very reason that they are only included to support Ada language features.

From the standpoint of the programming team (i.e., the users of an Ada compilation system), the effectiveness of the ACATS is measured by how well it covers the implicit parts of their application.

5. UNITS UNDER TEST

As a follow-on to the development of Apex for Linux, we decided to undertake an investigation into the effectiveness of the recently-completed tests of the compilation system. As a side benefit, the investigation would serve as a stress test of the testing tool's instrumented coverage feature.

With respect to the implicit code identified in section 3, above, the following units were identified for inclusion:

Predefined units – Units defined in LRM chapter 13 and Annexes A and B, with the exception of the following[1]:

- `Ada`
- `Ada.Unchecked_Conversion`
- `System` and child packages thereof
- `Standard`

Runtime units – Rational's full tasking runtime (handles all Ada95 tasking constructs, including task abort) for Linux POSIX threads

6. INVESTIGATION – REQUIREMENTS

6.1 Requirements Matrix

The requirements for any Ada compilation system are found in the Ada LRM (reference [4]).

At first, the task of analyzing requirements coverage by ACATS appeared to be trivial, since a coverage matrix tracing Ada requirements to the individual tests is included with the ACATS download.

Unfortunately, it turns out the coverage matrix is complete only for the features added by Ada 95. "Legacy" features that are common to the Ada 95 and Ada 83 dialects are generally not traced.

The effort to complete the coverage matrix by tracing the legacy requirements fell beyond the scope of this investigation.

6.2 Analysis

The coverage matrix provided with the ACATS is a simple text file. The traceability portion of the file was converted to a 2-column Excel spreadsheet showing each requirement (by individual paragraph number) in the first column, and the test or annotation for that requirement.

[1] These units are excluded because they cannot be instrumented without causing semantic problems.

In some cases the coverage matrix indicated that the requirement was only partially tested or had some other defect. These details were ignored.

A little sorting and filtering rendered the coverage matrix into a form where the tested vs. untested requirements could be tallied.

7. INVESTIGATION – STRUCTURAL

7.1 Tools

The investigation into structural coverage utilized the following:

- Rational Apex 4.2.2 for Linux
- Rational TestMate 4.2.2 for Linux
- ACATS 2.5 (with "J" modifications)

The two Rational products are the first Linux-hosted versions in the Apex product line. Both are included in, and marketed as, "IBM Rational Ada Developer Enterprise Edition" [5].

Rational Apex is a full life-cycle development system that provides Ada compilation in an integrated development environment. Rational TestMate is a testing system that includes test management, test execution, and coverage analysis in the Apex environment.

The ACATS version cited above was the most recent available at the time of the investigation.

7.2 Coverage Method

TestMate supports numerous coverage options, at both source code level (segment, decision, and MCDC) and object code level (segment and decision). Of the available options, source/segment was selected for several reasons.

- One of the simplest forms, easy to understand
- Option yields smallest number of coverage points
- Lower run-time overhead for very large numbers of coverage points
- Low set-up overhead because the first test's set-up is reused by subsequent tests

Coverage was obtained by inserting instrumentation points in the source code. The instrumentation points record subprogram entry and exit points, and each sequence of executable statements surrounded by decision points such as if statements.

7.3 Methodology

The investigation started with the creation of a full-source runtime, analogous to three full-source products we had created in the past for embedded variants. The source units for Apex 4.2.2's runtime were formed into a "full-source" implementation by the expedient of creating a package `rts_closure` that simply with's all the runtime units, and then with'ing that package in the Ada closure of the main unit.

In addition, we started with the full-source versions of the predefined units. (Rational normally ships the bodies as archive libraries which makes linking faster.)

At this point we had both the runtime and predefined units available in the Ada closure of the executable for each ACATS main unit. The runtime units were included through the specially constructed rts_closure unit, while the predefined units were included explicitly through the usual Ada "with" statements.

Next we made use of the "executable" tests from the ACATS – i.e., the Class C tests. Note that ACATS includes many semantic-only tests, such as those that only verify that the compiler detects semantic problems. Obviously, only those tests that lead to executable code are of interest to this investigation. The organization of the ACATS test suite is well-documented in reference [3].

These tests had previously been organized into TestMate-executable test lists and test cases. We ensured the inclusion of the predefined and runtime units by with'ing the `rts_closure` unit in the ACATS-defined package `report`. Because `report` is included in each of the hundreds of ACATS main units, this eliminated the need to modify each one individually.

At this point we ran the test lists three times, varying the options to TestMate. First, without the modified report package. Second, with the modified report package. And finally, with the modified report package, and with coverage collection enabled.

The purpose of the first run was to establish a baseline of functional results, and to verify that all the tests pass. The second test run was to verify that the predefined and runtime units could be included in this manner and pass all the tests. (The primary concern here was the potential for problems due to circular closures or elaboration difficulties, since this is not the way the runtimes are usually built.)

Finally, the third run gets to the heart of the matter – providing the structural coverage data that we are after. First, we verified the functional results – again, they all needed to pass. (And did.) Then we collected the TestMate-produced coverage reports.

7.4 Analysis

The resulting coverage reports were combined and then extracted into tabular form showing each coverage point, its location in the source code, and hit/not-hit data. This step yielded very large ASCII text files of over 10,000 lines each.

A few simple awk scripts were developed to process the tabular output and summarize the coverage points by subsystem, the unit of architectural control under Apex.

The subsystem data was easily mapped to predefined vs. runtime units.

8. RESULTS

8.1 Requirements Coverage

Table 1. Raw Requirements Analysis

Category	Raw		Adj.	
Tested	1999	*59%*	1999	*61%*
Nothing New (Ada83)	650	*19%*		
Not Tested	615	*18%*	1265	*39%*
Minimal Value	107	*3%*		
Total	***3371***		***3264***	

The raw analysis of the ACATS coverage file shows 3371 identified requirements (specifically, paragraphs of the LRM that contains a requirements) in total, of which about 22% fall into some category that is indeterminant -- not quite tested and not quite untested.

The "nothing new" paragraphs contain requirements that are unchanged in the Ada 95 update, and not traced to specific tests. The conservative adjustment for these tests is to count them as untested.

The "minimal value" paragraphs are testable but not meaningful. In this case, we will trust the initial judgment that went into the coverage matrix and count them as non-requirements.

The indeterminant cases are adjusted accordingly to produce the adjusted counts.

With adjusted scores, we end up with 61% requirements coverage.

8.2 Structural Coverage

Table 2 shows the raw number of coverage points executed (i.e., "hit") versus points that were not executed ("not hit").

Table 2. Raw Structural Analysis

Implicit Category	Units	Hit		Not Hit	
Predefined	477	2085	*24%*	6464	*76%*
Runtime	336	3076	*31%*	6634	*68%*

The final result is 24% structural coverage on a source/segment basis for predefined units, 31% for runtime units.

9. CONCLUSION AND DISCUSSION

9.1 ACATS Effectiveness

Figure 2 – Requirements

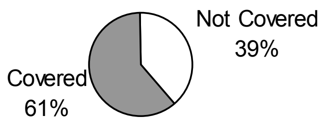
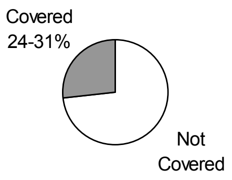


Figure 3 - Structural



The testing achieved 61% requirements coverage, and 24-31% structural coverage.

The conclusion of this investigation is that the ACATS is only moderately effective as a test suite.

9.2 Discussion

The ACATS comes out much better on the basis of requirements coverage than on the basis of structural coverage. This is not surprising, since the suite was originally developed as a comprehensive test of Ada compilation, based on the LRM.

Moreover, there are two factors that serve to understate the effectiveness of ACATS on a requirements basis. First, the Ada LRM is extremely detailed, much more so than a typical product specification. This serves makes coverage harder to obtain, but more profound when it succeeds. Second, the actual requirements coverage found is probably understated since many of the legacy (pre-Ada95) tests are not accounted for.

Also with respect to requirements analysis, there's a leap from the measure of requirements coverage for the ACATS as a whole to the implicit code of an application. No attempt has been made to determine which requirements are applicable to the predefined or runtime portions of an application versus those handled entirely by semantic analysis; such an analysis would require some sort of partitioning of the requirements.

With respect to structural coverage, the percentages achieved should be taken as approximate since the underlying code is implementation-dependent, depending on both the target machine and the vendor.

Several compiler vendors (Rational included) and customers have developed test suites with the goal of 100% runtime coverage for customized Ada subsets (e.g., non-tasking versions). This study suggests that these efforts are not misplaced, and may be useful as a supplement to ACATS.

9.3 Suggestions for Future Work

Analyze the older ACATS tests to attempt to fill in traceability of the Ada83 "legacy" requirements.

Extend the investigation to predefined units from the remaining LRM annexes.

Rule out "holes" in the ACATS. (Investigate the runtime units not covered by execution and determine if any major functions are untouched.) One specific hole we found in this effort is that there is no test of an enumeration representation clause with very sparse values.

Determine the effectiveness of Rational's other test suites as supplements to the ACATS.

10. REFERENCES

[1] Brukardt, R., Deller, S., and Tokar, J. Ada 95 Conformity Assessment. *ACM SIGAda Letters*, Vol. XIX, Issue 1 (March 1999), 52-57.

[2] Tonndorf, M. Ada Conformity Assessments: A Model for Other Programming Languages? *Proceedings of the ACM SIGAda Annual International Conference* (SIGAda 2003) (San Diego, CA, Dec. 2003). ACM, New York, NY, 10286.

[3] *The Ada Conformity Assessment Test Suite (ACATS) Version 2.5 User's Guide*. April 1, 2002. Ada Conformity Assessment Authority, Madison WI, April 1, 2002. This document and the test suite are available at http://www.adaic.org.

[4] *Ada 95 Reference Manual.* Intermetrics, Inc., Cambridge, MA, 1995. International Standard ISO/IEC 8652:1995.

[5] Product information is available at http://www.ibm.com.