Certified Testing of C Compilers for Embedded Systems

Olwen Morgan

Metriqa Ltd., 12, Maesderwenydd, Pencader, Carmarthenshire, UK SA39 9HF, omorgan@metriqa.com

Keywords: C compiler validation, certification testing

Abstract

For most embedded systems the implementation language of choice is C. Currently some vehicle-based software-controlled embedded systems, notably those for traction control and ABS, are assessed at ISO/IEC 61508 SIL4 [3]. At this level it is highly recommended that developers use certificated compilers. This paper sets out the rationale for such use, the nature of certification testing and what questions developers should ask their compiler vendors.

1 Introduction

Even today, automotive systems engineers are surprised at how many on-board computers modern vehicles have. The author knows of one top-of-range car audio system that contains no fewer than seven embedded microprocessors in its top-of range model. With that many for audio alone, it should surprise nobody that the microprocessor population of a modern saloon car can run to two dozen or more.

Vehicle-based embedded systems support applications at all safety-integrity levels. Audio systems are usually at SIL1 but an inertial sensing unit providing data for ABS and traction control is likely to be at SIL4. Even at SIL4, systems may be subject tight cost constraints. The target unit cost for an inertial sensing unit can be three dollars or less. Meeting high integrity levels at low cost requires a robust software process with dependable tools. Above all, it needs compilers to comply with language standards.

Historically, validation suites have been the only widely accepted tools for providing an assurance that a compiler complies with its language standard. The use of such suites stretches back over four decades 40. Salient examples (with approximate dates where known) over this period include:

- **Ada:** US DOD Ada Validation Suite (1980s)
- **C:** Plum-Hall Validation Suite for C, Perennial ACVS, CVSA and CVSA–Freestanding (1990 onwards)
- **C++:** Plum-Hall Validation suite for C++, Perennial C++VS and EC++VS (1990 onwards)
- **COBOL:** US Air Force COBOL Compiler Validation System (c1967), US Navy COBOL Audit Routines (c1968), US DOD COBOL Compiler Validation System (c1970)
- **Fortran:** NBS Fortran Test Programs (c1973)
- **Java:** Plum-Hall Validation Suite for Java, Perennial JETS (1998 onwards)
- **Pascal:** Pascal Compiler validation Suite (1981)

As this (non-exhaustive) list shows, compiler validation suites have been developed for most widely used programming languages very soon after they have been standardised. An interesting account of such developments up to around 1979 is given in [2]. In addition to these suites, various ad-hoc tests have been developed and placed in the public domain, notably Kahan's `paranoia` tool for testing arithmetic and mathematical functions. Originally developed for BASIC implementations, it is now available for Fortran, Pascal, Ada, C, C++ and Java.

Most of the test suites have been used in formal compiler validation services offered in the US by NBS and in the UK by BSI. European standards and certification bodies also offered services in their own countries and protocols for mutual recognition of validation certificates exist.

The availability of test suites and services nevertheless belies the technical difficulties of establishing and sustaining a reliable test capability. Like any other form of testing, compiler testing needs to produce accurate, repeatable and reproducible results. This requires careful test suite design and well controlled test procedures. Few software engineers in the world have experience in both of these areas. Therein lie many pitfalls for unwary test suite developers, test service providers and customers for test reports.

The remainder of this paper discusses the essentials of test suite design, the nature of controlled testing procedures and issues facing providers and users of testing services. The special aspects of testing implementations of floating-point arithmetic are covered and the paper concludes with a checklist of advice for compiler vendors, testers and users.

2 Essentials of test suite design

The body of knowledge in test suite design is sparsely documented – possibly owing to the specialised nature of the techniques used. Wichmann and Sale, the developers of the Pascal Compiler Validation Suite (PCVS), sought from the outset to follow key principles of metrology in their design, which they documented in [5]. Though long out of print, this book is essential reading for anyone designing a serious compiler test suite. Key facets of their design were:

- **Checksumming the code** of test programs so that their correct conveyance on media or via data networks could be verified.
- **Testing only one requirement per program**. This yields a test suite with a large number of small programs, which is serendipitously convenient for testing cross compilers for small target systems.
- **Classification of tests.** Test programs in PVCS were divided several classes of which the main ones were:
  - *CONFORMANCE*: These programs were always in correct standard language and were expected to compile and achieve a "PASS" verdict when run
  - *DEVIANCE:* These programs were never in standard language but differed from it in some subtle way. They detected compilers that supported language extensions, failed to check or limit some standard language feature or exhibited some common error,
  - *IMPLEMENTATION-DEFINED:* These programs class were always in standard language but used implementation-defined features. They tested the compiler's handling of such features and when run, either diagnosed the nature of the feature of produced a verdict of "NOT DETERMINED".
  - *IMPLEMENTATION-DEPENDENT:* Programs in this class sought to determine how implementation-dependent features were provided. They were not necessarily all in standard language and could yield indeterminate verdicts.
  - *ERROR-HANDLING:* These tests were designed to evoke *exactly* one error. Each such program was preceded by a *pre-test program*, which was as nearly identical to it as possible but did not contain constructs that evoked the error. Pre-tests were all in standard language and checked that the actual test did not fail for reasons unrelated to the specific error for which the compiler's behaviour was being tested.

For the five main classes, analogous classes of tests can be defined for C. The three classes of *CONFORMANCE*, *DEVIANCE* and *IMPLEMENTATION-DEFINED* are conceptually identical to those for Pascal. Pascal's *IMPLEMENTATION-DEPENDENT* class corresponds approximately to tests of unspecified features in C while, again approximately, the *ERROR-HANDLING* class corresponds to tests of what behaviour the implementation provides in cases where the standard leaves the behaviour undefined.

The identification of these test classes was an important development since each class requires a subtly different kind of test design. More importantly, systematic test classification allows the design of automated test management software to be simplified yet permits more insightful interpretation of test results. As a technique of suite design it can be applied to any language. Unfortunately this lesson has been lost on some developers of subsequent test suites.

- **Portability:** All tests, even those of implementation-defined features, were designed to be portable, i.e. to deliver reliable verdicts for all conforming compilers. This was a significant advance on previous test suite designs. COBOL and Fortran tests had often needed manual adaptation to run with some compilers.
- **Support for automatic processing of results:** The first version of PCVS had around 400 programs. Later versions reached nearer to 800. This was due to the "one requirement per program" approach. It needed automatic tools to gather and format results later printing. To facilitate this, outputs from test programs followed a standard format that could be easily processed by test management software.
- **Separation of syntax and semantics:** Testing of syntactic and semantic features was separated. Hence, for example, the tests for accepting all the syntactic forms of logical expressions were distinct from those used to test the evaluation of Boolean operators.

In the first decade of the 21st century, these approaches seem eminent good sense. So indeed they were regarded in 1981 when the PCVS was first published. Individually all of them can be seen traced to earlier test suites. What Wichmann and Sale did for the first time, however, was to apply *all* of these principles in a single suite. In so doing, they set a technical standard that developers of subsequent test suites often chose to ignore and their work remains a valuable reference for anyone seeking to evaluate test suites for C.

3 Controlling test procedures

Compiler testing must deliver accurate, repeatable and reproducible results. Without careful test suite design, none of these qualities is attainable. But though good suite design is necessary, it is by no means sufficient. Without controlled test procedures, all of them can be prejudiced.

Ideally a compiler testing organisation should offer a service complying with ISO/IEC 17025 [4] which essentially means that all test methods used must be technically fit for purpose. To make compiler testing procedures fit for purpose is not as technically straightforward as it may seem.

3.1 Accuracy

Technical measures supporting accuracy are deployed mostly in the design of test programs. An obvious example for C is how to determine whether plain `char` is signed or unsigned. Writing a conforming *portable* program to do this is by no means as easy as it may at first appear. As regards accuracy, the unambiguous diagnosis of implementation-defined and unspecified features makes the greatest technical demands on the test suite designer.

Nevertheless, without sound test procedures, even the best-designed tests may not produce accurate answers. As regards the signedness of plain `char`, many C compilers provide a compile-time option to select signed or unsigned treatment. A robust test procedure must record exactly which options are used. The simplest way to do this is never to put such options on the command line but gather them in a file that is referenced from the command line. The contents of the file can then be reproduced as an accurate record of what the compile-time options were during the test.

Hence, accuracy relies in no small degree on the design of test management software.

3.2 Repeatability and reproducibility

Repeatability and reproducibility are often confused. Testing is *repeatable* when successive runs of tests on the same equipment by the same tester produce results that match each other according to defined criteria. It is *reproducible* when runs of tests on different but technically similar equipment, and carried out by persons other than the original tester, produce results that match the original results according to defined criteria of agreement. Roughly speaking repeatability indicates good control of local testing procedures while reproducibility indicates technically coherence across different test equipments and procedures.

The key to repeatability is to ensure total control of all factors that may influence the results of tests. Metriqa compiler test systems, for example, are **never** connected to an external network so that no special measures are needed to prevent network traffic from interfering with test operation. The procedures are also designed to establish a tightly controlled directory environment from which tests are retrieved and compiled and to which test results are stored as they are produced. Configuration checks run before and after each test ensure that the state of the configuration is as expected throughout any given period of testing. This also supports restarting of failed tests if required and verification of result transcripts at very fine levels of granularity.

Metriqa test systems also provide support for reproducibility. In a recent validation, our equipment was unplugged from one machine, plugged into another and tests rerun producing identical results. This can be achieved between any two systems that run the same compiler and support mounting of externally connected disk drives as host system directories. Metriqa can currently do this using USB and SCSI connected test configurations. Our most compact USB-connected test configuration is so small that we can fit several into a cabin-baggage-sized case.

3.3 Configuration control

To summarise, the procedural side of accuracy, repeatability and reproducibility is best assured by design of automated test procedures that perform continuous configuration checks to:

1. Control all factors, static and dynamic that may affect the results of tests,
2. Check invariants that should hold at key waypoints in correctly controlled tests,
3. Record the results of all such verifications as the test session progresses.

As far as the author is aware, Metriqa is the first organisation to research this area systematically and our methods remain under development (and for the moment under wraps).

4 Special issues for embedded targets

Testing of C compilers for embedded systems poses technical difficulties not encountered in hosted environments. Users of test suites need to be aware of these difficulties and how test suites deal with them.

4.1 Test suites for embedded C implementations

In the terminology of the C standard, a cross-compiler provides a freestanding implementation and the only libraries that standard requires it to support are:

`<float.h>` `<iso646.h>` `<limits.h>`, `<stdarg.h>`, `<stdbool.h>`, `<stddef,h>` and `<stdint.h>`.

Conspicuously missing from this list are `<math.h>`, whose functions are frequently required in control applications, and `<errno.h>`, which `<math.h>` typically requires for exception values. Also absent is `<string.h>`, whose simpler functions are often useful in embedded applications. More significantly, for testing embedded targets, no I/O facilities are required.

Given these frugally limited requirements for freestanding implementations, the question of the scope of a test suite arises. Providers of C compiler test suites such as Plum-Hall and Perennial both offer reduced-scope test suites for such implementations. Since most embedded compiler vendors offer an embedded `<stdio.h>` with limited basic facilities, both Plum-Hall and Perennial are able to use them to create output from test programs. Otherwise, both are able to use I/O routines customised for environment that do not rely on `<stdio.h>` functions. The comments below refer to the freestanding versions of both suites.

In their cut-down forms for freestanding implementations, neither suite tests mathematical functions particularly thoroughly and tests of floating-point arithmetic are less extensive than those provided by some other test tools.

The Plum-Hall test suite does not follow the one-requirement-one-test approach and consists of just over 200 test programs. In contrast, the Perennial suite sticks to one-requirement-one-test and contains over 8000 small programs.

Small program size means that Perennial test object codes are smaller than Plum-Hall ones and may run in embedded environments with limited memory that are too small for some Plum-Hall tests. By testing several requirements in a single program, the Plum-Hall tests also risk producing inaccurate results if failures early in a program prejudice the results of later tests in the same program. As regards isolation of test failures, therefore, the Perennial suite is the better designed of the two.

4.2 Aspects of test suite design

Curiously, neither vendor of C test suites appears yet to supply checksums with which to check the integrity of the test suite code as delivered to customers. Of more concern, however, is that at least one of these suites uses conditional compilation so that the source code actually compiled depends on the testing of conditions within the translation environment by the pre-processor. Effectively this lets the implementation under test decide for itself what tests it will be subjected to. The author views this as undesirable and advises clients against using suites that make unnecessary use of conditional compilation. Another undesirable aspect is that some tests of unspecified or undefined features do not have corresponding pre-tests. This too is poor test design and to be avoided as far as is reasonably practical.

4.3 Running tests

Both Plum-Hall and Perennial test suites come with scripts to automate testing, but these may need significant adaptation for embedded implementations. As far as the author is aware, they do not contain the kinds of features that Metriqa test systems use in support of repeatability and reproducibility. Moreover, most users testing embedded C compilers have developed their own test management software, usually with ergonomic rather metrological consideration in mind.

Running tests on slower embedded targets can be time-consuming. Elapsed time to run a test suite is dominated, however, not by processor speed but by the time to download test program object images from the host to the target. This in turn depends on the characteristics of the (usually) USB host-target transfer interface. Table 1 gives approximate times observed by Metriqa for a recent series of 215 tests.

Target	Elapsed time
M16 C	2.5 hrs
M32C	1.5
V850	45 minutes
ARM 9	15 minutes

Table 1: Elapsed time for tests

Running all 8000+ programs of the Perennial suite would on this basis take about 10 hours for the ARM 9 target and 100 hours for the M16C. A 10-hour elapsed time is manageable but 100 hours would span over two working weeks. This is one reason why automated test management software needs to support suspend and restart facilities.

Another option, of course, would be to connect several identical target processors to the host. Two 4-port USB interfaces, for example, would support eight externally connected targets reducing M16C elapsed time to the order of 12 hours. Special controls are needed when using multiple targets but they do not pose any new technical difficulties.

5 Testing floating-point characteristics

5.1 What is special about floating-point?

Many coding standards ban floating-point arithmetic (FPA) in critical applications. Why this restriction for something that amounts to just doing sums? One reason is that few software engineers have the skills required for coding of sensitive numerical processes. Even in seemingly simple cases detailed knowledge of numerical methods may be required to ensure that processes are robust and stable.

Another reason is that FPA has often been implemented incorrectly, not only in software but also in hardware. (In one case a well-known trademark was irreverently claimed to stand for "<u>P</u>roduces <u>E</u>rroneous <u>N</u>umbers <u>T</u>hrough <u>I</u>ncorrect <u>U</u>nderstanding of <u>M</u>athematics"). In critical applications, one should be suspicious of any floating-point implementation until there is evidence that positively suggests its correctness.

5.2 How should floating-point be tested?

The question in this section's title is easily answered: *by experts, with exceptional care and using specialised tests*. Users requiring such tests need to supplement validation suites with more specialised test tools.

Tests of floating-point characteristics are of two kinds:

(a) checks on parameters of floating-point representations and the accuracy of basic arithmetic operators,

(b) checks on the accuracy of the mathematical functions.

For (a) the methods of Cody and Waite are indispensable. The public-domain tool **`paranoia`** uses them and both it and **`esparanoia`**, a version adapted for embedded systems are freely available. Users should appreciate, however, that these tools depart from the one-requirement-one test principle and may give misleading results under adverse conditions. That reservation notwithstanding, both tools are useful in testing floating-point characteristics. If either produces results that do not agree with **`<float.h>`**, it is a significant reason to doubt the quality of the floating-point implementation.

Testing the accuracy of mathematical functions is a much harder task. The approach used in the Pascal Compiler Validation Suite was to produce reference implementations of each function in standard language using only floating-point types and their arithmetic operations. The results produced by such implementations were, as far as possible, accurate to the last bit of their representations.

To get that accuracy one uses a good rational approximation then a fast-converging iteration to refine it. Cody and Waite [1] give examples in Fortran but production of C reference implementations is not a matter of mere translation. It helps to be conversant with Chebyshev approximation, Lipschitz conditions and methods for superquadratic convergence, so volunteers should form an orderly queue! ….

6 Compiler testing services

6.1 Lessons of economic history

Historically, few compiler validation services have been commercially viable. The only one that survived for any length of time was that for Ada where use of the name "Ada" was permitted only to validated compilers. The past decade, however, has seen Ada displaced by C and largely through economic forces. The widespread availability of embedded C implementations made it the only viable choice when software needed to be developed for multiple targets.

In earlier validation services the vendor bore the cost of validation, submitted the compiler for testing and specified the options under which it was claimed to conform to the standard. The tester then carried out the tests and produced the validation report. While this model was accepted for compilers of Pascal and Ada, it is far from ideal for embedded C compilers, where different applications may require radically differing invocation options.

Costs were high. Most compilers were tested only under one set of options and testing was repeated annually or on major new releases. In a market with a limited number of vendors, the total market for testing was so small that charges were dominated by overheads and some testing operations never broke even on a commercial basis.

6.2 New validation services

Under the vendor-funded business model, even small testing organisations with low overheads cannot today make adequate revenues at less than around $20,000 per validation. Nevertheless change is in the wind. ISO/IEC 61508, and sector-specific standards based on it are creating a new demand for validation of embedded C compilers. This demand is eliciting new business models for compiler testing.

Metriqa has a user-funded business model. We test at a vendor's site and then provide validation reports to users at a cost comparable with that of an extra seat for the compiler. Reports are valid only for the user's licensed compiler and continued validity requires maintenance payments in step with the software licence. The user gets updated test reports for any new releases during the period of a report's validity.

Other business models are also emerging as testers enter the market and users can expect steady evolution of commercial validation services under competitive market conditions.

6.3 Who watches over the testers?

As new testing services emerge with diverse business models, users need assurance about the quality of those services. In the UK, any testing organisation that is serious about its work should have or be seeking to achieve UKAS accreditation as a testing laboratory. The author is not aware of any testing organisations that currently hold UKAS accreditation specifically in the compiler-testing field but it is an area in which users are wise to monitor developments.

7 A short checklist for compiler users

With the market in a period of rapid development, what should users look for in C compiler validation? The following non-exhaustive checklist identifies issues to raise with vendors and testers:

1. The tester should be independent of the vendor and the business arrangements between them should not create incentives that prejudice the tester's impartiality.
2. The tester should use recognised test suites (for C either Plum-Hall or Perennial) and or tests that are in the public domain. Test reports should state which version of a suite was used for testing.
3. The tester should use test management software that provides adequate controls in support of accuracy, repeatability and reproducibility.
4. Test reports should state only conclusions that are supported by the results of testing.
5. Reports should state any qualifications and limitations that apply to the reported tests. In particular a statement of test conditions should clearly identify the options used at compiler invocations.
6. Reports should be signed by the tester and second person to certify that they have been conducted as stated. **They do not certify the compiler. They certify that the testing described in the report was carried out as described.**
7. Supply of test reports should be under written contract, which should clearly state the liability of the tester in respect of the tests.

These items of advice are for guidance only and should not be taken as exhaustive. Nevertheless, enquiries based on items 3 and 4 may cause red faces among some testers.

Acknowledgements

The author gratefully acknowledges the opportunity afforded to her by IAR in connection with the development of test procedures for embedded C compilers.

References

[1] Cody, W. J., and Waite, W., Software Manual for the Elementary Functions, Prentice-Hall, 1980, ISBN-10: 0138220646.

[2] Dwyer, D. J. and Noble, D. I., (eds.), *Language Implementation Validation*, National Computing Centre Ltd., April 1980

[3] IEC 61508-3 *Functional safety of electrical / electronic / programmable electronic safety-related systems – Part 3: Software requirements*

[4] ISO/IEC 17025:2005 *General requirements for the competence of calibration and testing laboratories*

[5] Wichmann, B. A. and Ciechanowicz, Z. J., *Pascal Compiler Validation*, Wiley, 1983, ISBN-10: 0-471-90133-4