

The Anatomy of an FAA-Qualifiable Ada Subset Compiler

V. Santhanam
The Boeing Company
Wichita Development & Modification
Center
1-316-523-2014
vdot.Santhanam@boeing.com

ABSTRACT

To date, compilers used in the construction of FAA-certified software have not been specially qualified for use in safety-critical applications. This has resulted in costly test procedures that attempt to verify the correctness and achieve structural coverage of the object code program rather than the source program. The FAA has adapted the criteria set forth in the document RTCA DO-178B for the design, development, and verification of tools used in the construction of safety-critical software. Tools meeting those criteria can be qualified as code development tools, and their output can be used without additional verification. This paper describes the internals of a compiler for a subset of Ada that is designed to meet the qualification criteria. Use of the qualified compiler will reduce the test burden for certifying flight software at the highest levels of criticality.

Categories and Subject Descriptors

D.3.4 [Software; Programming Languages; Processors];
D.2.4 [Software/Program Verification];
D.4.7 [Organization and Design].

General Terms: Standardization, Languages, Verification.

Keywords: Software certification; FAA; DO-178B; Compiler; Ada.

1. INTRODUCTION

The Federal Aviation Administration levies stringent testing requirements on safety critical software. Various levels of criticality and corresponding rigor of testing are outlined in the document RTCA DO-178B [1], which is the current basis of certification for virtually all commercial flight software. For example, level A represents the highest level of safety criticality, and software at this level must be tested to demonstrate structural coverage, including statement, decision, condition¹, and modified

¹ In DO-178B terminology, a decision is a Boolean expression consisting of one or more Boolean terms, known as conditions, combined by Boolean operators such as AND, OR.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGAda '02, December 8–12, 2002, Houston, Texas, USA.
Copyright 2002 ACM 1-58113-611-0/02/0012...\$5.00.

condition-decision coverage² (MCDC). All levels of criticality require functional (requirements-based) testing. The structural coverage requirement ensures that any functionality that remains untested by functional tests is subsequently tested.

Ultimately, the software whose correctness is the subject of all testing is the sequence of instructions that resides in the embedded processor. That software is rarely produced directly by programmers. In practice, the programmers write source code in some high order language (HOL), which is compiled by a compiler. The output of the compiler may be binary object code, or an assembly language program that must be assembled into binary object. Object code modules produced by separate compilations must then be linked to form an executable. That executable must be loaded into the embedded processor's main memory before the software becomes functional. Thus, we could have a compiler, an assembler, a linker, and a loader, all performing transformations on the source code to produce the functional software. Through all of these transformations, the integrity of the software must be maintained. It is expensive and, perhaps even impractical, to manually verify the output of each of these tools and be assured that the software originally written in HOL has not changed its character due to an erroneous transformation by that tool.

DO-178B levies the same testing requirement for any tool that replaces a manual step in the creation of the embedded software as is levied on the software itself. The process is called qualification, to distinguish it from certification, which applies to the embedded software. Thus, if a flight control system is written in Ada and targeted for level A certification, the Ada compiler, the assembler (if one is used), the linker, and the loader must all be qualified as level A tools or else the output of each unqualified tool must be independently verified. This paper describes a tool suite that is designed to meet the FAA criteria for qualification as a level A code development tool.

2. QUALIFYING A COMPILER

What does it mean to qualify a compiler tool suite per DO-178B requirements? First of all, the objectives of qualification must be established. Informally, the objective is to eliminate the need to verify the application software at the object code level. To achieve this, the object code generated by the compiler must have the following properties:

² MCDC is way of exercising complex Boolean decisions to demonstrate the independence of the conditions involved. See [2] for an elaborate discussion of MCDC.

- ◆ Must be deterministic,
- ◆ Must have direct, context-free mapping to the source code statements,
- ◆ Must not contain code that will never be executed (dead code),
- ◆ Must not contain code that represents functionality beyond the semantics of the source code construct.

These properties, which help establish the correctness of the object code, often run counter to the objectives of a commercial compiler. For example, generating object code that maps to source code without regards to the context in which the construct appears, could rule out many of the traditional optimizations involving code motion performed by a typical commercial compiler. The need to avoid dead code may make it necessary in some instances to generate calls to library functions rather than inline code. The need to avoid extraneous functionality may preclude the use of special instructions to support debugging.

Secondly, the tool suite must be developed following a rigorous software development process that includes requirement definition phase, a detailed design phase, an implementation phase, and a verification phase. The flow-down of the requirements must be explicitly documented. For example, the design elements must refer to the requirements they address, the implementation must refer to the derived requirements of the design elements, and the tests must refer to the low-level requirements they help verify. Most commercial compilers may not have this documentation trail, making it necessary to reverse engineer the artifacts necessary to demonstrate that a sound software engineering process was followed in the creation of the tools.

Finally, the tools must be verified to the stringent standards applicable to a software development tool. Per DO-178B, these are the same standards that are applicable to the embedded software itself and the level to which it is to be certified. This means that to qualify the compiler suite to level A would require that each tool in the suite be tested to demonstrate statement, decision, and MCDC coverage. Let us consider each tool in the compiler-assembler-linker-loader chain individually to assess the level of effort needed.

The simplest of the tools in the chain is the loader. Typically, it does not perform any transformation of the executable, but it needs to maintain the integrity of the executable as it transfers it from an external medium to the main memory of the embedded processor. In our experience, such a tool can be constructed in a straightforward manner in a few hundred lines of source code, and can be readily tested to DO-178B level A standards. Furthermore, using suitable checksums, the integrity of the loaded software can be verified to a high-level of confidence.

The assembler and the linker, likewise, are not very complex. Our experience shows that each of these can be constructed in under 10,000 lines of Ada code, leading to moderate testing costs for level A qualification.

Qualifying a compiler for a high-order language such as (full) Ada can be daunting. For one, the transformation induced by a compiler is far more complex than that induced by an assembler or a linker. Marketplace pressure to produce the tightest possible object code only adds to the complexity. The compiler itself can easily be two orders of magnitude larger than an assembler or

linker. For example, the GNAT compiler³ for Ada 95 consists of about 625,000 lines of Ada and C code. We have estimated that level A structural coverage testing of the GNAT compiler will require 58 person years of effort⁴. The total cost for tool qualification will likely be 3-5 times the cost of structural coverage testing. The cost of qualifying other COTS compilers for comparable languages will likely be equally monumental, which is probably why there are no currently FAA-qualified compilers in existence to our knowledge.

Since not all the complexity of full Ada is needed in embedded applications, it makes sense to seek to reduce the complexity of the compiler by identifying a subset of Ada for its input language. We have identified such a subset for our compiler that we describe below.

3. THE SUBSET

The selection of the language subset was recognized at the outset to be crucial to the success of the project. We began by pruning major features of Ada that are not absolutely necessary for embedded applications based on a single-threaded, cyclic-executive design. Accordingly, we dropped tasking. We then dropped generics in favor of a separate macro preprocessor, a `la` `cpp`, if necessary. We also decided to leave out tagged types and associated object oriented features, primarily because there were no clear guidelines for certification of software based on the OO paradigm at the time. Finally, all forms of dynamic allocation were eliminated, as most embedded systems voluntarily avoid using dynamic memory due to the risk of memory fragmentation and leaks. As a consequence, certain other constructs whose translation depends on dynamic memory management at run time, such as functions returning unconstrained type results and non-static aggregates, were also outlawed.

Besides these major exclusions, we restricted the language in a few places where the compiler complexity did not seem to justify the programming convenience. Overloading, child/subunits, and derived types are some of the features that were outlawed or restricted. The only form of overloading permitted in the subset is the inherent overloading of predefined operators; users may not define operator functions or other subprograms that overload another declaration. Our experience showed that the packages of Ada serve quite well to modularize the design and development of Ada software; child units and subunits contribute little to enhance this feature. Finally, we nipped derived types and the complex semantics of derived subprograms, but retained the ability to introduce new scalar types based on predefined types, a corner stone of Ada's strong typing.

A guiding principle in the selection of the subset was the need to remain a strict subset (both syntactically and semantically) of Ada. That would allow us to develop our software using COTS software engineering environments (e.g., GLIDE by ACT), eliminate the need to develop a whole new SEE for The subset.

Another driver was the need to keep the mapping between source statements and object code as direct as possible. To this end, we

³ Based on source code in GNAT version 3.13p.

⁴ Using an estimating tool for Ada, and projections for C based on our experience.

restricted the semantics of certain constructs, mostly dealing with operations on composite types. For example, the comparison of two record objects containing holes was disallowed.

Of course, it was important to ensure the subset was powerful enough to be useful in implementing embedded applications. Our target was small to medium sized applications (under 100,000 lines of source code) based on cyclic executive design paradigm (i.e., single-threaded). We used a number of embedded avionics applications that had been certified by the FAA using DO-178B as the basis. We analyzed all the constructs used in those applications and the frequency with which they were used. The most frequently used features were automatically added to the subset unless they violated the previously stated principles. The less frequently used features were then analyzed to determine if they could be readily recoded using features already included in the subset. An infrequently used feature was added to the subset only if it was determined to be essential for safety-critical, embedded programming.

Some of the larger applications we studied were implemented in Ada 83. It was necessary to evaluate even the frequently used features to determine if they were needed in our Ada 95 subset. For example, a large application we studied used subprogram renamings quite lavishly. A closer look at those uses determined that the coding pattern was necessary to facilitate operator visibility, and the *use type* clause of the subset would have obviated most of them.

4. THE ARCHITECTURE

In many ways, our compiler architecture is not unlike any multi-

pass compiler targeting to a virtual machine, as depicted in figure 1. The design and implementation of these traditional components, however, are quite untraditional.

4.1 The Virtual Machine

We chose to target our compiler to a virtual machine (VM) rather than a specific COTS processor for a number of reasons. A suitably designed virtual machine will greatly simplify code generation and make it easy to demonstrate the simple source-to-object mapping that is called for in DO-178B. Secondly, a virtual machine will pave a smooth migration path for the application software. If and when it becomes necessary to move the application to a newer platform, the virtual machine is the only component that needs to be ported (and re-certified); the legacy certification artifacts of the application software are carried over to the new platform unaffected. Thirdly, the virtual machine provides new opportunities to collect run-time statistics (e.g., structural coverage data, branch history, exception log, etc.), which would have been impossible without changes to the software under a conventional compiler and run-time system.

There is, of course, a performance penalty for the use of virtual machine. Based on our prototype, this penalty is estimated to be a performance reduction by a factor of 10 to 25 compared to native code implementation. While this may seem like a large penalty, the cost and speed of today's processors makes it possible to implement applications under a VM that would have required native code implementation on processors from just a couple of years ago.

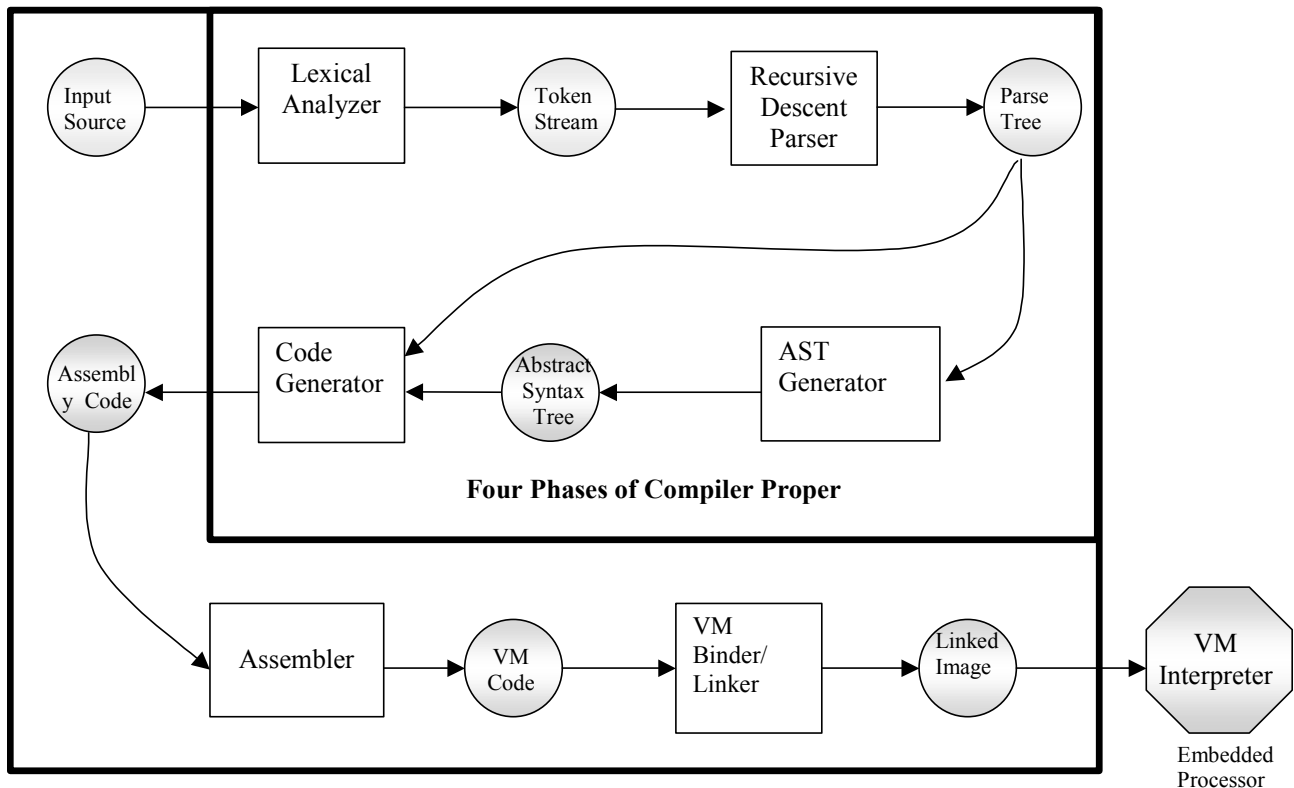


Figure 1. High-Level Architecture of Compiler Suite

4.2 The Compiler

The single, most-important driver of the design of the compiler was the need to qualify it as a DO-178B level A software development tool. That meant testability. The attribute that has the most direct impact on testability of any software is its size. We decided that the subset compiler should be no more than 50,000 SLOC. The next most significant attribute is the testability of individual subprograms. We use an estimating tool, described in [5], to determine a testability score that is a composite of the statements, branching constructs (IF/CASE), loops, complex Boolean decisions, and number of other subprograms called. A score of n translates to $n/2$ hours to test the subprogram. We have kept the structure of 90% of the subprograms in the compiler to under a composite score of 50, and 99% under 100.

We have used a small, portable subset of Ada, not unlike the input language itself, for the implementation of the compiler. While there are some advantages to implementing the compiler in its own subset [3], we found that the addition of limited forms of generics and dynamic memory allocation was necessary to keep the compiler code straightforward (after all a compiler is not an embedded application and we would not expect the subset designed for embedded applications to adequately meet a compiler's needs). Portability was important so that we could use multiple compilers to demonstrate the integrity of the compiler at the object code level. Compiling our compiler using a second Ada compiler and demonstrating identical results will add to the confidence that the primary Ada compiler used to construct our subset compiler did not generate faulty code.

Our code generator is table-driven, thanks to the virtual machine architecture that closely matches the features of the subset. The code generator makes no serious attempt to optimize the code generator. Much of the optimization that is performed can be classified as peephole optimization, such as selecting a byte constant load instead of a word constant load when the constant is small enough to fit within a byte. There are no loops ever generated for a non-looping construct, such as long operand compares or assignments; such operations are either directly supported in the virtual machine or they are forbidden in the subset. When an error is detected in the input program, the compiler will put out an error message and identify the portion of input text in which the error was identified. There is no serious attempt to repair the input to either avoid collateral diagnostics or continue processing the remaining input.

In coding the compiler, we have employed a number of defensive programming techniques in order to build greater confidence in its correctness.

- ◆ We use assertions liberally. Assertion-based programming is not only a great debugging aid, but it also assists with maintenance and formal proof where applicable [4].
- ◆ The compiler itself is compiled with no optimizations turned on in order to minimize chances of erroneous code generated for the compiler and to ensure a more direct object-to-source mapping.
- ◆ None of the language-required run-time checks are suppressed in the compilation of the subset compiler.
- ◆ We avoid nested subprograms and packages. This simplifies module testing at the subprogram level, if that becomes necessary to meet structural coverage criteria.

5. VERIFICATION OF THE COMPILER

To verify the compiler, we have chosen a multi-pronged approach. First, we selected all those tests from Ada Compiler Validation Suite (ACVC) 2.1 that were applicable to the subset; there were 562 tests that were determined to be applicable. Some rewriting of most of these tests was necessary to replace incidental uses of restricted features, e.g., elimination of package use clauses, un-nesting nested subprograms, and replacing functions returning unconstrained result types.

The ACVC tests were complemented with compiler-specific tests. About half of these tests were aimed at confirming that the subset restrictions are correctly enforced by the compiler. The remaining tests were used to exercise the portions of the compiler that were not exercised by the other tests. As of the writing of this paper, the compiler specific test suite contained 116 tests.

We have also assembled a suite of "regression tests" arising from user-reported bugs in the compiler. This is a strategy that other compiler vendors have employed effectively to ensure that their compilers do not regress as new fixes are put in. Our regression suite consists of 104 tests at the present time.

A number of additional verification tasks are in the planning. To verify that the compiler suite itself is compiled correctly, the suite will be recompiled using a second compiler and all tests will be rerun. The suite's portability (specifically the portability of the virtual machine interpreter) will be demonstrated by running all executable tests (ones that compile, link, and run) on both a little-endian processor and a big-endian processor.

6. CONCLUSION

We have demonstrated that it is possible to construct an FAA-qualifiable compiler per all levels of DO-178B for a modest subset of Ada. Such a compiler will facilitate the construction of safety-critical software that is tested and validated entirely at the source code level, leading to significant savings in the certification of such software. We view the compiler as complementary to COTS compilers that are well integrated with powerful development aids including language-sensitive editors and symbolic debuggers.

7. REFERENCES

- [1] *DO-178B: Software Consideration in Airborne Systems and Equipment Certification*, RTCA, Inc., 1992.
- [2] Chilenski, J. J.; Miller, S. P. Applicability of modified condition/decision coverage to software testing, *Software Engineering Journal* v.9, n.5, Sept. 1994.
- [3] Goerigk, Wolfgang. On Trojan Horses in Compiler Implementations, Technical Report, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, Germany, 1998.
- [4] Bates, Rodney M. Debugging with assertions, *C/C++ Users' Journal* 10, Oct. 1992.
- [5] Santhanam, Usha. Automating Software Module Testing for FAA Certification, *Proc. SIGAda 2001*, p.31-37, Sep. 2001.
- [6] Amy, Peter. A language for systems not just software, *Proc. SIGAda 2001*, p.3-11, Sep. 2001.