Safety Certification of Compiler Generated Object Code

INTRODUCTION

This paper examines the standards and guidance related to safety certification of object code generated by a compiler toolchain for both CPU and GPU targets. It is written in the context of two markets: avionics (DO-178C/ED-12C) and automotive (ISO 26262). (ISO 26262 is derived from the general IEC 61508, which is also used and derived for other markets with safety-critical applications, such as rail and nuclear). This paper will explain the motivation behind the guidance and identify approaches that may be used to address concerns. Understanding the guidance and constraints will facilitate the selection of an appropriate approach.

In the context of this paper, a compiler is a computer program that translates a high-level programming language, such as C, GLSL, or SPIR-V, to a lower-level language, such as object code, that can be used to create an executable program. See Figure 1.

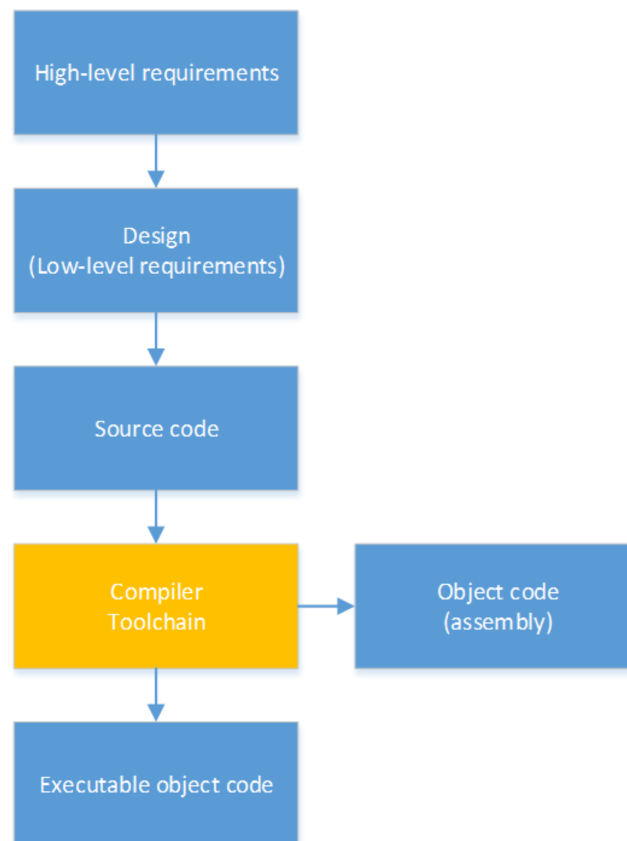

Figure 1: Compiler Tool Chain Relative to Development

Compilers perform many or all the following operations: preprocessing, lexical analysis, parsing, semantic analysis, code optimization, and code generation. Typically, compilers have an option to produce an intermediate assembly language output that is a human-readable representation of the executable object code (one to one relationship). Compilers are complex computer programs consisting of millions of lines of code; they are a concern for safety-critical development in that they are tools that generate code embedded in the application, and any error in the generated executable object code can introduce a failure in the safety-critical application. Industry standards and guidelines, therefore, include guidance on developing sufficient trust in the compiler-generated executable object code.

This paper assumes the application is developed to the highest safety criticality level, DO-178C level A and ISO 26262 ASIL D.

AVIONICS CONTEXT (DO-178C/ED-12C)

For avionics safety-critical applications, the intended function is captured through functional and safety requirements. There is an emphasis on reviews and requirements-based verification to ensure the code meets requirements and source-code-to-object-code traceability analysis, which ensures that added code—not directly traceable to the source code—is appropriate and correct (including any added library function calls). This is shown in Figure 2.

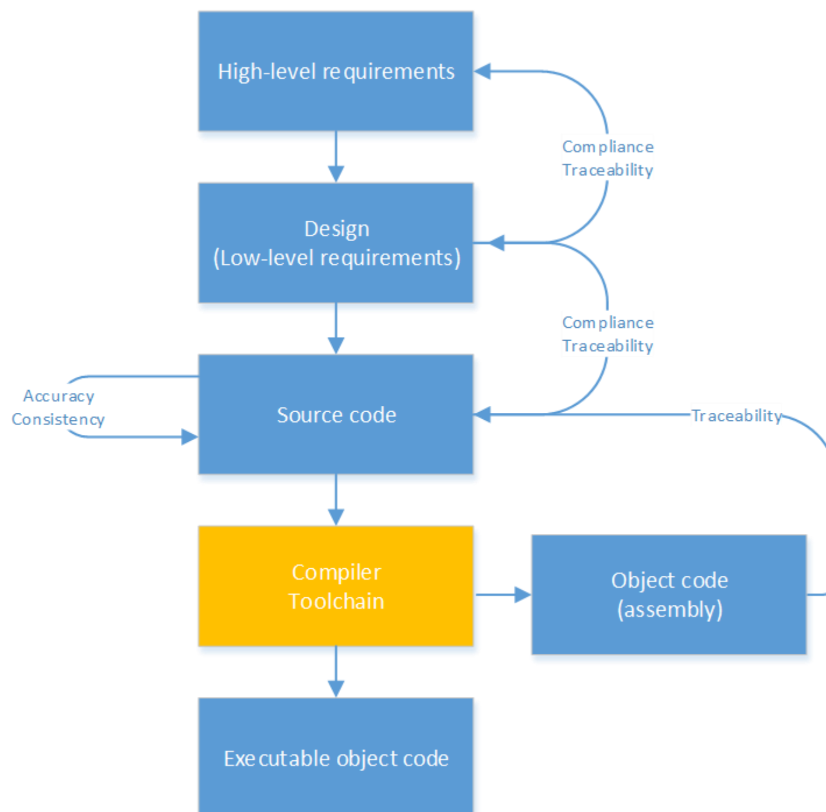

Figure 2: Source Code to Object Code Traceability

According to DO-178C subsection 6.4.4.2.b, the objective of source-code-to-object-code traceability analysis is to verify all additional non-traceable executable object code. Source-code-to-object-code traceability, while part of the structural coverage analysis objective, is a supplemental objective. Examples of non-traceable object code include when the compiler adds initialization, error-detection, exception handling, hidden library routines, and, in the case of an object-oriented programming language, data structures and code for the object-oriented features. Unfortunately, non-traceable object code is not always identifiable to a tool, meaning some manual analysis may be necessary. After the additional code sequences are identified, their functionality is analyzed to determine if it is appropriate and correct for the functionality associated with the code from which it was generated. Generally, this additional verification is done to determine acceptability.

Position paper CAST 12 covers source-code-to-object-code traceability and identifies two analysis approaches: analyze the complete application object code, or analyze a complete set of used/implemented programming constructs (representative code). Representative source code is developed by writing code samples for every language feature permitted by the coding standard, the restrictions being used, and sometimes for the most complex aspect of the application. The code samples are compiled using the same compiler and options as the application, and the generated assembly code is analyzed for added code. The process of analyzing representative code is depicted in Figure 3.

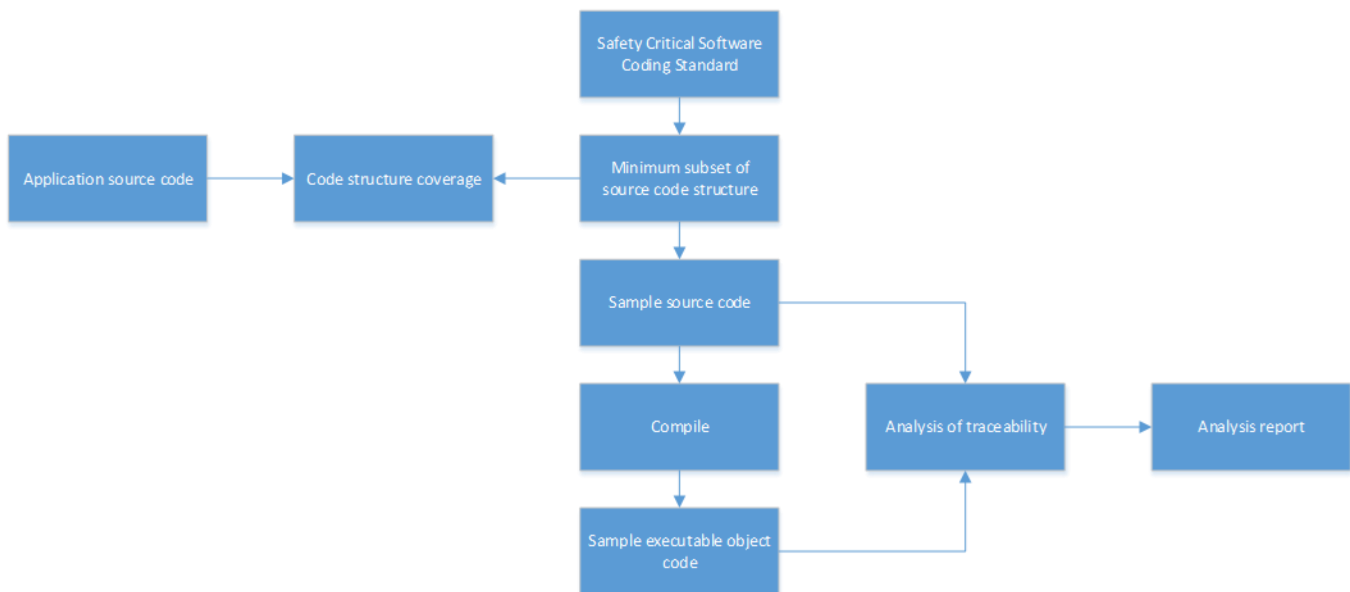

Figure 3: Analysis of Representative Code

This analysis focuses on the compiler-added code—ignoring the rest of the code—because it implements the source code that is verified by requirements-based testing. Compilers may optimize for performance and rearrange the object code such that one-to-one correspondence between source code statements and sequential object code blocks are not maintained. This also includes the specialized case when compilers for parallel SIMD units transform the code for parallel execution. This transformation does not change the end function. DO-178C subsection 4.4.2.a states that if test cases provide coverage consistent with the software level, the correctness of the optimization does not need to

be verified. For example, if MC/DC analysis of the source code achieves coverage, there are no additional activities because of optimization at the object code level.

While analysis of structural coverage can reveal shortcomings in requirements-based test cases, inadequacies in requirements, dead code, deactivated code, or unintended functionality, it would not identify branches in compiler-added code that are not executed in requirements-based tests. Therefore, source-code-to-object-code traceability analysis is needed in addition to source code coverage analysis.

While structural coverage is traditionally performed on the source code, it may also be performed on object code. Some considerations for selecting structural coverage of object code are:

1. It can support more valid coverage as testing and coverage analysis are conducted on an abstraction of the code that is closer to the final airborne software than the source code. However, the mapping of the object code to source code decisions will need to be analyzed to show that the test suite exercises all the source code (compiler optimization makes it difficult to be certain that all source code has a test case), which may result in additional MC/DC testing.
2. Structural coverage analysis tools exist and have advanced to support this approach. This is true when targeting a CPU, but it is currently not the case when targeting a GPU, for which manual analysis would need to be performed.
3. It may not require instrumentation, which is the case for many CPU targets for which built-in trace hardware is available; however, this is not the case for GPU targets.
4. It is required for SEAL (Safety Evidence Assurance Level) certification. SEAL process is defined in the Joint Strike Fighter (JSF) System Software Development Plan (AS SDP), and is based on DO-178B, DEFSTAN 0056, and MIL-STD-882.

The level of compiler optimization complicates the analysis of object code because the logical abstractions at the source code level might not be present at the object code level. This usually results in restrictive coding standards and limitations on the compiler such that code complexity and compiler optimization is limited to enable object code coverage.

Upon successful completion of verification of the software product, the compiler is considered acceptable for that product provided the test cases give coverage consistent with the software level, and the non-directly traceable object code is addressed (per DO-178C subsection 4.4.2). Therefore, a DO-330 qualified compiler is normally not necessary.

AUTOMOTIVE CONTEXT (ISO 26262)

ISO 26262 is the main functional safety standard concerning system, hardware, and software development for safety-critical in-vehicle applications. It defines requirements and constraints for development, validation, and verification activities. ISO 26262:6 addresses software, while ISO 26262:8 subsection 11 addresses confidence in the use of software tools.

We will start with determining the need for compiler qualification per ISO 26262:8 subsection 11, given that the

compiler-generated executable object code is embedded in the application and any error in this code can introduce a failure in the safety-critical application. This results in the Tool Impact (TI) being classified as TI2.

Following the example provided by ISO 26262, the Tool error Detection (TD) classification of TD1 is selected for a compiler (code generator) when the generated code is verified following ISO 26262 (ISO 26262:8 subsection 11.4.5.2, example 1). That is, there is a high degree of confidence that a malfunction and its corresponding erroneous output will be prevented or detected. This is similar to DO-178C avionics guidance in that the following activities are performed on the software:

1. Requirements based tests run on the executable object code, inclusive of robustness testing
2. Special focus on boundary values, etc.
3. High-level code MC/DC coverage analysis
4. Source-code-to-object-code analysis (analyzing compiler-added code, such as complex libraries)

With a Tool Impact classification of TI2, Tool error Detection of TD1, and following ISO:26262 Tool Confidence Level (TCL) determination, we have a classification of TCL1 and no compiler qualification is needed.

Refer to the *Compiler Errors Application Note*, that examines potential compiler errors.

SUMMARY

In this paper we have described the guidelines for safety certification of compiler-generated code for both avionics (DO-178C) and automotive (ISO 26262). The emphasis on avionics reflects its long history and that a similar approach is adopted for both.

Provided the generated executable object code is verified following the industry guidance for the assurance level, the compiler is accepted, meaning there is high confidence in the output for the verified application. For avionics and automotive, the verification includes both normal and robustness test cases that are requirements-based, as well as MC/DC coverage analysis. An additional requirement in the acceptance of compiler-generated object code is the source-code-to-object-code traceability analysis, used to uncover and address compiler-added code.

A qualified DO-330 TQL-1 or ISO 26262:11 TCL-3 compiler would allow you to skip the source to object code analysis; however, at the time of writing, no one has been able to do this for avionics applications. Otherwise, there are no formal certification credits in exchange for tool qualification offered by these industry standards. That is, it does not reduce verification or source-code-to-object-code verification activities.

Structural coverage may be performed on either source code or object code. If performed on source code, the compiler-added code sequences in the object code need to be identified, functionality analyzed, and analyzed for appropriateness and correctness. This may be performed on representative code rather than 100% of the generated object code. If structural coverage is performed on the object code, you will need to analyze the mapping of object code to source code

decisions for completeness.

While CPU targets are well supported by tools to automate both source code and object code structural coverage analysis, GPU targets can have some tool support for source code structural coverage analysis automation, with object code structural analysis being primarily a manual activity.

CoreAVI supports GPU code generation using an offline GLSL compiler toolchain provided with our drivers and libraries, such as CoreAVI's ArgusCore™ SC 2.0, VkCore® SC, and VkCoreGL® SC2.

AUTHOR

Gregory Sikkens
Director, Safety Solutions Architect

Gregory Sikkens has over 30 years of experience holding a wide range of technical and management positions within the rugged embedded COTS market. Prior to joining CoreAVI, Gregory was the senior product manager at Curtiss-Wright responsible for directing and launching a number of advanced graphics and FAA DO-254 certifiable COTS modules. Furthermore, Gregory has extensive hands-on experience in developing and managing OpenGL SC graphics driver development including DO-178 DAL A software development.

The information contained in this document is for informational purposes only and is subject to change without notice. CoreAVI, the CoreAVI tracer logo, and combinations thereof are trademarks of CoreAVI. All other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

© 2020 Core Avionics & Industrial Inc. All rights reserved.