The Benefits of C and C++ Compiler Qualification

Abstract

In embedded application development, the correct operation of the compilation toolset is critical to the functional safety of the application. Two options are available to build trust in the correct operation of the compiler: either by compiler qualification through testing, or application coverage testing at the machine code level. We argue that the first, compiler qualification, is much more efficient. In addition, separating compiler qualification from application development shortens the critical path to application deployment (time-to-market) because they are then independent of each other. Compiler qualification saves time and money.

Functional Safety standards such as ISO 26262 for the automotive industry describe if and how tools, such as software compilers, must be qualified if they are used in safety critical applications. Tool *Qualification* is the process that is described by a functional safety standard to develop sufficient confidence in using a tool.

Compilers are complex pieces of software (in the order of 2 to 5 million lines of code) that play a crucial role in the translation from source code to the machine code. That machine code becomes part of the safety critical application or device. Any error in the generated code can introduce an arbitrary safety critical event.

To fulfill the essential requirement that the compiler makes no error in code generation, ISO 26262 requires an important choice to be made: it is necessary to *either* develop sufficient trust in the compiler through qualification, *or* develop application test procedures that are strong enough to detect any error in the generated machine code. In this paper, it is shown that the choice for compiler qualification is the more efficient one.

Let us first explore the actions required if the compiler is not qualified. The compiler does not need to be qualified if there is a "*high degree of confidence that a malfunction [of the compiler] and its corresponding erroneous output will be prevented or detected*" (ISO 26262, Part 8, 11.4.5.2.b.1). We argue that this comes at a high price. To build that confidence it is necessary to test the application *on-target* and to analyze its coverage (statement, branch, MC/DC) at the *machine code level*.

On-target means that the tests that are developed for the application must be tested on the actual target processor hardware. Implied is that the compiler is *in-the-loop* of the test procedures.

In Part 6 (product software) Table 12, the ISO 26262 standard requires a specific level of coverage analysis as part of unit testing, depending on the ASIL level. This can be statement, branch and/or MC/DC analysis. For integration testing, function and call coverage testing are added in Table 15.

If the compiler cannot be trusted and in the presence of compiler optimizations, this analysis must be performed at the level of machine code and not at the level of source code. It is not sufficient to do coverage and branch analysis on the source code. The reason for this is that compiler optimizations significantly transform application code. As a result, even with 100% source code coverage, many branches and code blocks are likely not covered at machine code level. Therefore, with source code coverage only there is not a *high degree of confidence that a malfunction [of the compiler] will be prevented or detected*.

In the Appendix, a simple function with a loop is analyzed. It demonstrates how large the gap is between source code coverage and coverage at the machine code level.

Without compiler qualification, the only way to gain confidence that a malfunction in the compiler is detected is by demonstrating sufficient coverage of code and branches at machine code level.

Fine Print: Section 9.4.5, Note 4

In Part 6 (product software), Section 9.4.5, Note 4, the ISO 26262 standard states that software unit testing (in this case: coverage analysis) can be carried out at the source code level, followed by "back-to-back" testing of the unit tests. "Back-to-back" testing means that the results of on-target test-runs are compared to the results of *on-host* (emulated or simulated) test-runs.

This note in the standard states a requirement on application software testing. It can be used to argue that the *application* software is sufficiently tested and that it works on the actual hardware. But for the reason discussed in the previous section, it does not imply that the *compiler* can be trusted.

The Benefits of Compiler Qualification

Now alternatively, consider application testing with a qualified compiler. Compiler qualification, for example by testing against the compiler specification, is the process that is described by the Functional Safety standard to gain sufficient confidence in the correctness of the compiler. It is independent of the application that is being developed, but depends on the "use case" of the compiler: how the compiler is used to compile the application. For example, this includes the specific option settings and optimization level of the compiler.

With a qualified compiler, the application developer can trust that malfunctions of the compiler are detected in the qualification process. This means that a compiler does not have to be free of defects (few compilers are), but that the defects are known to the

application developer so they can be avoided.

With a qualified compiler, application testing does not have to take into account the artifacts introduced by the compiler. The compiler can be trusted, so it does not have to be in-the-loop of the coverage testing process. Coverage testing can proceed at source code level. (This is so for ISO 26262. For DO-178C, additional measures are required.)

The benefit of a trusted compiler is that it significantly simplifies the test procedures for the application and makes them more efficient. It is true that compiler qualification itself is not trivial, but it is a process that can be managed in-house without much overhead.

Here are some advantages:

- Compiler qualification is done only when a new compiler (update) is introduced. That is at most two to three times per year and is not on the critical path of application development. Without compiler testing, application testing at machine code level is on the critical path to deployment and happens every time the application is updated. Thus, the path to deployment (time-to-market) is shortened when a qualified compiler is used.

- If a single application is compiled with multiple compilers and is deployed on many target platforms, qualified compilers significantly reduce the need for on-target testing. As stated above, back-to-back testing is still needed but it does not have to consider the actions of the compiler. With trusted compilers, application testing can focus on source code validation.

- Writing application unit tests to cover code and branches that were generated by the compiler at machine code level makes these tests compiler dependent. The same tests may not be sufficient to cover generated code by another compiler, or even an update of the existing compiler. When the compiler is trusted, coverage analysis can focus on source code coverage, which is independent of the target compiler and platform. No compiler specific unit tests are needed.

- Compilers perform more transformations to the code at higher optimization levels. More transformations make it harder to write coverage tests. This may be a reason to lower the optimization level of the compiler used, which may lead to reduced performance and higher resource usage by the application. Therefore, a qualified compiler, at the highest desired optimization level, can lead to more efficient application code and a reduction of the required hardware resources.

- It raises the confidence that the crucial step of generating machine code from source code is done correctly. This is true in particular when the qualification of the compiler is done with the same compiler option combination that is used in the deployment of the application.

Clearly the greatest win from using a qualified compiler is that application testing can focus on the application source code and not on artifacts introduced by the compiler. This is important for application developers because they want to work on the correctness of their application and not on the correctness of the tools they use. By

separating the concern for the application from the concern for the compilation tools, it becomes easier and more efficient to develop the application and deploy it on multiple targets.

Compiler Qualification for Compiler Users

Compiler qualification is best done by the application developer, instead of the compiler supplier, because the qualification must match the application use case of the compiler as closely as possible. Given that every compiler has a near infinite combination of options, it is unlikely that the compiler supplier has tested against the actual compiler options that the application developer uses.

Compiler qualification needs to be set up carefully, but it is not a hugely challenging process. All it takes are the guidelines of the ISO 26262 standard, or another Functional Safety standard, and a compiler test suite that is grounded on the compiler/language specification, such as SuperTest for C and C++. With these, an automated qualification process can be set up that is easy to repeat for different compilers, and for updates of existing compilers.

Summary

As a compiler user, you cannot rely on your compiler supplier to qualify the compiler for your specific use case in a functional safety critical domain. Nor can you rely on application testing to prove the compiler correct. Separating application testing from compiler (tool) qualification makes application deployment more efficient and hence, more cost effective. At Solid Sands, we are happy to guide you with the compiler qualification process.

Appendix
Coverage Analysis and Compiler Optimizations

The following is a simple function with one loop:

```
int f (int n) {
     int total = 0;
     for (int i = 0 ; i < n ; i++) {
          total += i & n;
     }
     return total;
}
```

When interpreted at the source code level, this code contains a single conditional branch based on the condition '**i < n**'. A unit test for this function that calls it with a non-zero positive value for the argument '**n**' will completely cover all statements in the function. Moreover, this single unit test will trigger the conditional branch in both ways: at least once to enter the loop body, and once to exit the loop. As a consequence, a single unit test suffices for MC/DC coverage of the loop at source code level.

When the code is compiled (in this case with an LLVM based x86 compiler) with optimization level **-O0**, it is translated more or less literally into machine code. Although there is no guarantee that this is always the case (another argument in favor of compiler qualification), inspection shows that the same unit test as the above also provides full MC/DC coverage of the machine code.

However, at level **-O0** the compiler does not perform register allocation and all variable manipulation is done on the stack. The resulting code is both run-time inefficient and not compact. At the very least, one would want to compile at level **-O1**.

At optimization level **-O1**, the code is much more compact, runs three times faster and still resembles the source code sufficiently well to be compared manually. Nonetheless, the compiler did introduce an additional branch statement that provides a shortcut if the loop body is never executed (when the variable '**n**' equals zero). This case is not handled by the unit test. An additional unit test must be created that calls the function with an argument value zero. Otherwise, there is no complete branch coverage at the machine code level.

For another factor six (!) in performance, the code is compiled at level **-O2**. The compiler performs loop unrolling and vectorization. The resulting machine code is incomprehensible at first glance. It is still incomprehensible at a second glance. It consists of thirteen basic blocks and has nine conditional branches. Not only are the two unit tests completely inadequate to achieve full code and branch coverage, we also do not have an easy means to create sufficient unit tests. The reason is that we do not have tools at our disposal to perform satisfactory MC/DC analysis at machine code level. Many tools for MC/DC analysis at source code exist. Only few, and for few processors, exist for analysis at machine code level.

Does one need to use the compiler at a higher optimization level than **-O0**? Without optimization, the target code is close to the source code structure and the original tests for full source code coverage suffice. The answer is that it depends on the application. In this simple example, there is a factor eighteen difference in performance between **-O0** and **-O2**. That translates into a factor eighteen more resource usage, including power. In many application areas, it will not be acceptable to leave such an improvement on the table.