Using Open-Source Compilers in Safety Critical Projects

Dr. M. Beemster, CTO, Solid Sands B.V., marcel@solidsands.nl

Abstract

One of the great aspects of functional safety standards is that they do not care about the origin of a tool that is used for a safety critical application. Focusing on compilers in particular, it does not matter if it is bought from a reputable supplier or downloaded from GitHub. What does matter is that the process for qualification as described in the functional safety standard is followed. C and C++ compilers are tools. The ISO 26262 standard, and similar standards such as IEC 61508, have different requirements for tools than for the software that is developed for the application itself. ISO 26262 offers four different paths to tool qualification. Out of these four, the path of qualification by "Validation of the compiler" is the most practical and common. Validation shows compliance of the compiler to its specification. This is performed by using a test suite that is based on the C and C++ language definition. Luckily, and unlike most other programming languages, the C and C++ languages have well defined ISO specifications. This paper explores compiler qualification for safety-critical applications with an emphasis on open-source compilers.

Functional Safety

The goal of functional safety is to reduce the risk of harm to people to an acceptable level in the case of a hazard or failure. That is a lofty goal, but it does not tell us what to do to achieve safety. It also leaves a lot to be defined. Also note that it does not mention open-source software and with further refinement of this goal into functional safety standards this remains true. Functional safety does not depend on software being open source or not.

The refinement of our functional safety goal is embodied by functional safety standards. These standards are rooted in expertise gained in this area for over 150 years (since the industrial revolution). Instead of prescribing how systems shall be designed (which would stifle innovation and, over time, miss the point) they start from safety requirements that leave freedom in how they are achieved. For specific application areas, further refinements of these requirements exist in the form of standards that define safety processes.

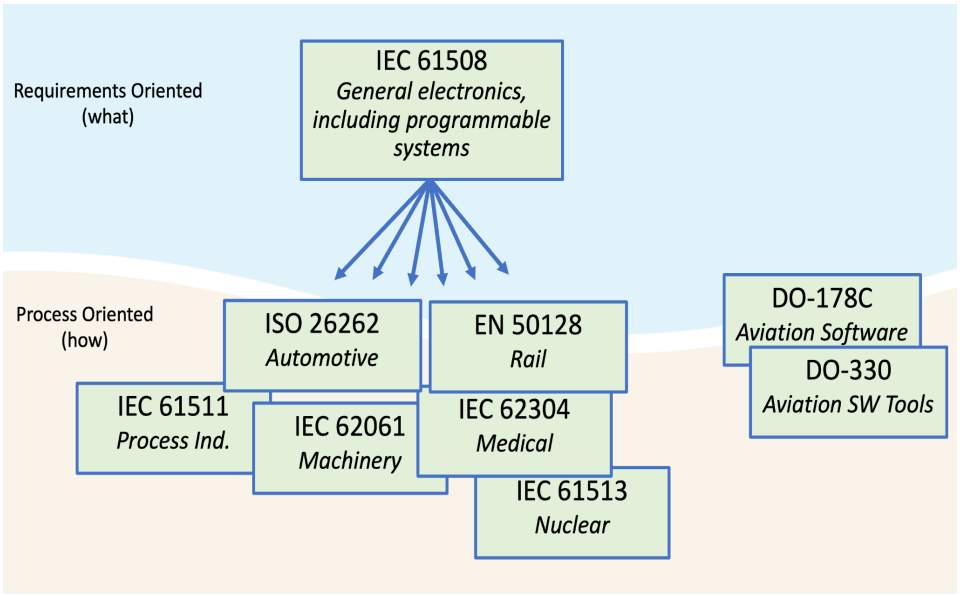

Figure 1: Functional safety standards in the domains of electronics and software

In the past ten years, cars have turned into software defined vehicles. For this reason, safety of software has become one of the primary pillars of car safety. Figure 1 lists a number of functional safety standards that deal with software. On the left side is the family of IEC 61508 related standards, on the right side two aviation related standards, one for general aviation software, the second for tools such as compilers. The standards below IEC 61508 are refinements of it for specific domains, with a key difference: they are more *how-to* oriented, providing a specified process. Instead, the IEC 61508 standard from which they are derived states the requirements for safety-critical systems, but does not prescribe how those requirements are to be met. As a result, the more process-oriented standards are easier to work with because they tell you what to do. However, if that process does not fit with a specific situation, one can fall back on the requirements of IEC 61508.

Focusing on compilers specifically, there is a difference between their treatment in the IEC 61508 family and the aviation standards. Compilers are (off-line) tools. They are the most safety-critical tool in the arsenal of the application developer, but they are not embedded into the target application. They are also deterministic and have a clear and simple input-output relation. This makes them less critical than the software that goes into, for example, the brakes control software. For that reason, IEC 61508 allows the qualification of compilers by treating them as a black box. One has to verify that the output of the compiler is compliant with its specification, but it is not required to dive into the inner workings of the compiler. The difference with the aviation standards is that they do not treat the compiler as a black box. This has important implications for open-source compilers, as we will see later.

Evaluation of the Risk of a Compiler

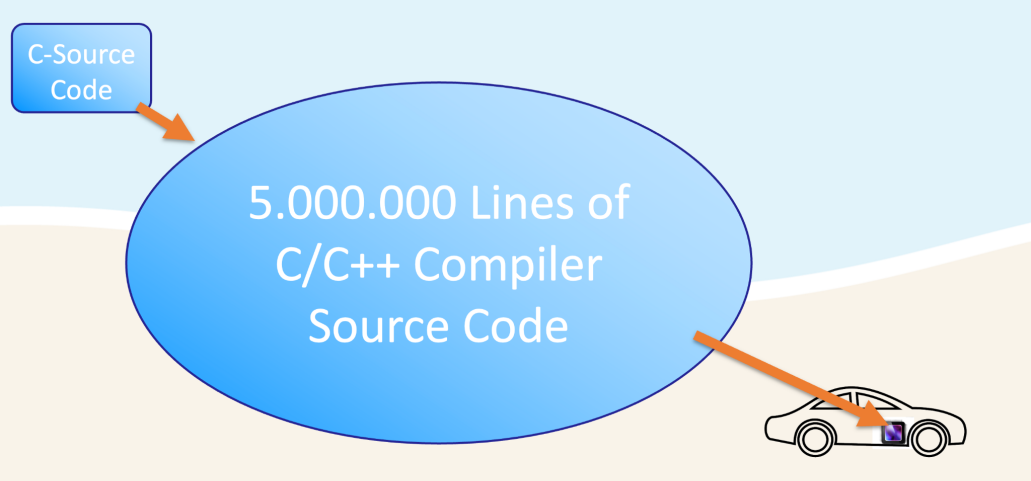

Figure 2: Compiler complexity dwarfs the complexity of most safety-critical software in a car

Despite the black-box approach to compiler qualification, compiler complexity, and by implication their ability to insert defects into the application, cannot be underestimated. The number of lines in modern open-source compilers such as Clang and Gcc is in the order of five million. That is significantly more than the number of lines in most safety-critical systems in cars. Open-source compilers are used for many safety-critical applications. Yet, their development spans more than 40 years (for Gcc), they have thousands of contributors, they are not developed with functional-safety standards in mind, they are not compliant to guidelines such as MISRA, and they do not have a test set that comes even remotely close to full statement or branch coverage. These are all significant risk-factors.

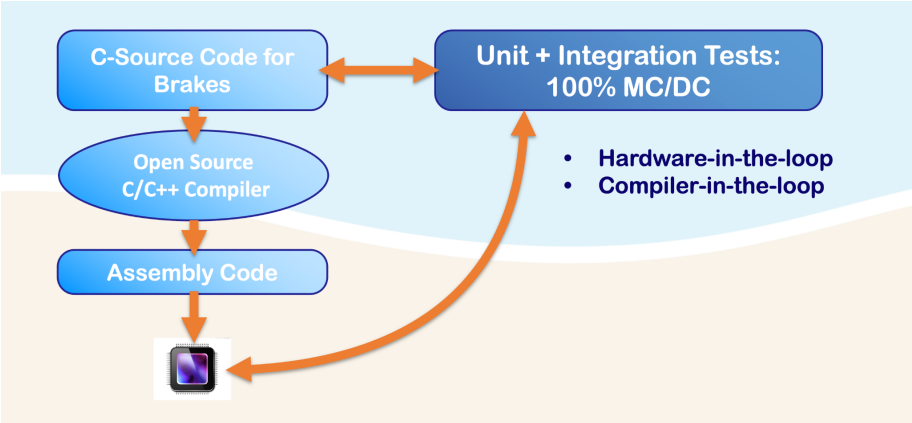

Figure 3: Application testing with compiler and hardware in the loop

For these and other reasons such as the complexity of the specification, compiler qualification cannot be taken lightly. ISO 26262 does not require that every tool used for application development is qualified. If it can be shown not to pose a safety risk, its qualification can be skipped (see Tool Confidence levels in ISO 26262) .

An important criterion here is the question if, in the case of a compiler defect, its occurrence is detected by other verification techniques. A potential argument in favor of this is sketched in Figure 3. In this figure, on the left side is the application that is compiled for and executed on a target processor. The block on the right side represents the collection of unit and integration tests for the application. These tests are executed with the hardware, and by extension the compiler, in the loop. As required for ASIL D (ISO 26262's highest level), the tests should show full statement and branch coverage of the source code.

Is this setup good enough to ensure that all potential compiler defects are found? ISO 26262 states that one has to be conservative with this assessment and, given the complexity of the compiler and its role in the development, by default the answer is no.

The following example gives one argument why this is the case.

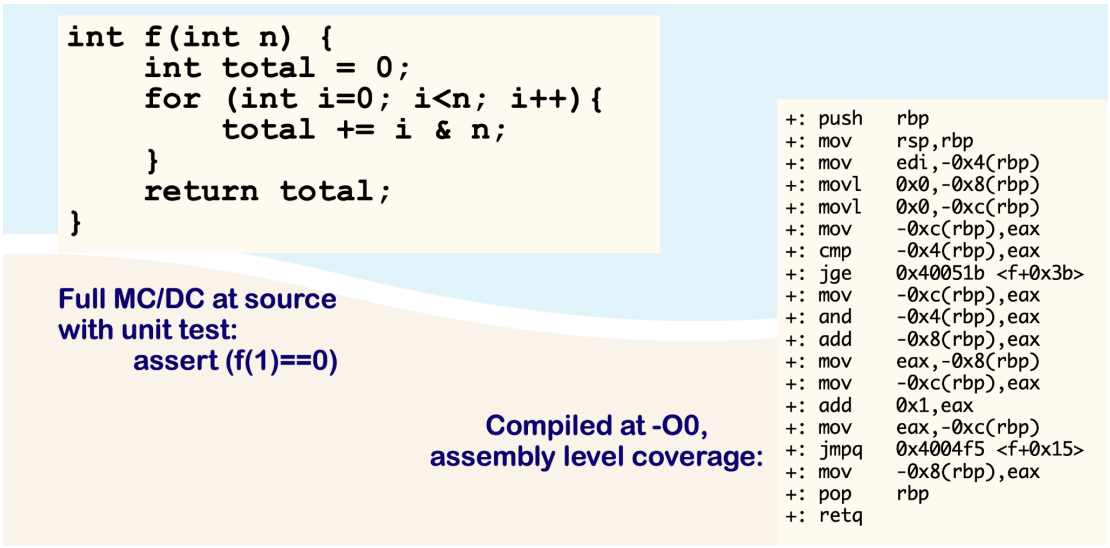

Figure 4: A simple loop compiled without optimization

Figure 4, in the left upper corner, shows a simple but not completely trivial function. This code has complete statement and branch coverage at the source code level when the function is called with **`f(1)`**. Granted, that is not a very thorough test, but it is sufficient to pass the code coverage requirement. At the right side of the picture is the assembly code generated by the compiler when the function is compiled without optimizations. The plusses in front of every instruction show that we have full coverage also at this level, with the same test case. This includes the conditional branch instruction (**`jge`**), which only gets a plus if it is called both in the taken and non-taken direction.

So far so good, but compiling without optimization generates extremely inefficient code. This may be acceptable in some cases, but for mass-market applications it can be extremely costly. Optimized code can easily be three times faster than not optimized code and for computation-oriented code it can be closer to ten times faster because of auto-vectorization. A factor three means that for the same performance one can use a three times slower processor, which may be three times cheaper, which requires three times less power, requires less cooling and so has smaller packaging. This turns compiler optimization into tangible cost savings.

Let's turn on the optimizer.

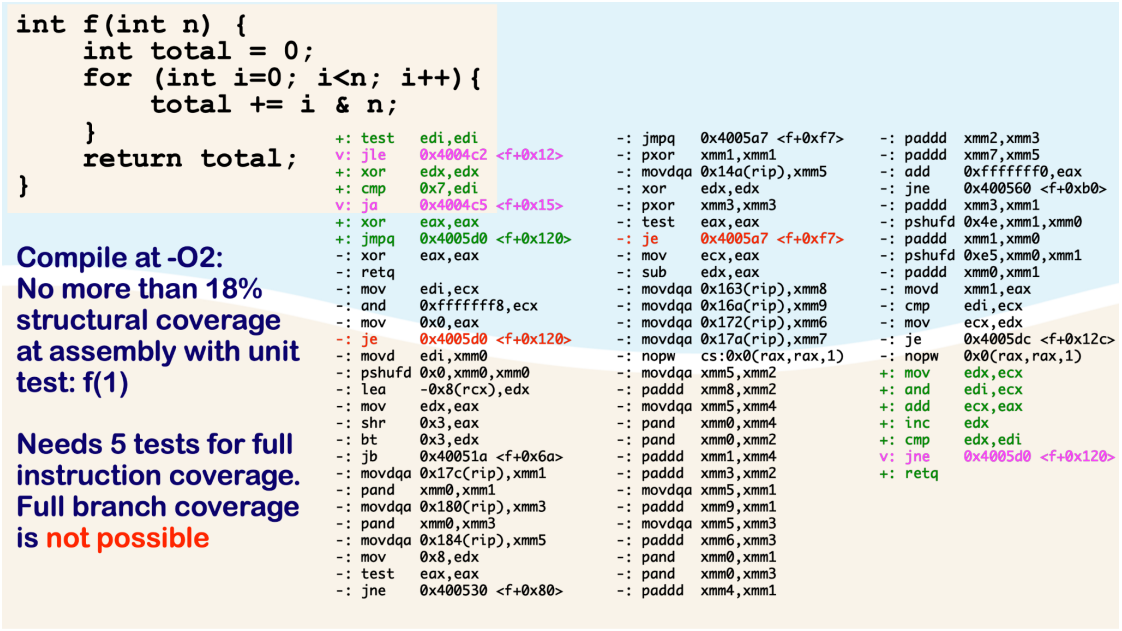

Figure 5: A simple loop compiled with -O2 level optimization, with assembly code coverage

Figure 5 shows the same function, but now compiled with optimization level -O2. This is the level at which the Linux kernel is compiled by default, which makes it relevant. The number of assembly instructions has increased significantly. This is because the loop is unrolled and vectorized. At the instruction level, only the green instructions are fully covered by the test expression **`f(1)`**. It is just a very small percentage of all instructions. The pink instructions are conditional branches that are taken in one direction but not both. Although **`f(1)`** is not a representative test for the loop, this example highlights that a trivial test can be sufficient for source code coverage, but wholly unsuited for assembly code coverage.

Taking one step back, we should ask if there is a risk to having untested instructions, let's call them *dark code*, in the target application. If we have full coverage at the source code level, is there any risk to the application containing dark code that may never be executed? The answer to this is clearly yes, there is a risk. There is no guarantee or indication whatsoever that this code will not be executed in the actual application. It just means that the precise calls of the function to trigger the dark code were not part of the test set. The compiler has generated this code to deal with specific cases. If left untested, we cannot assume that the assembly code is correct, which is the code running on the final hardware.

To achieve maximal instruction and branch coverage for this example, at least four additional, precisely targeted function calls are needed. Even with these additional tests, full branch coverage cannot be achieved. Analysis shows that the two conditional jumps marked in red are only ever taken in one way. These branch instructions are inserted by the compiler to take care of a specific case, but they are shadowed by a previous handling of that same case. For example, the red instruction **`je 0x4005d0`** on the left side and the pink instruction **`ja 0x4004c5`** both compare the argument **`n`** to seven, although in a different way. In theory, a further optimization of the code could clean up the redundancy of the second conditional branch, but there is no requirement that the compiler does that.

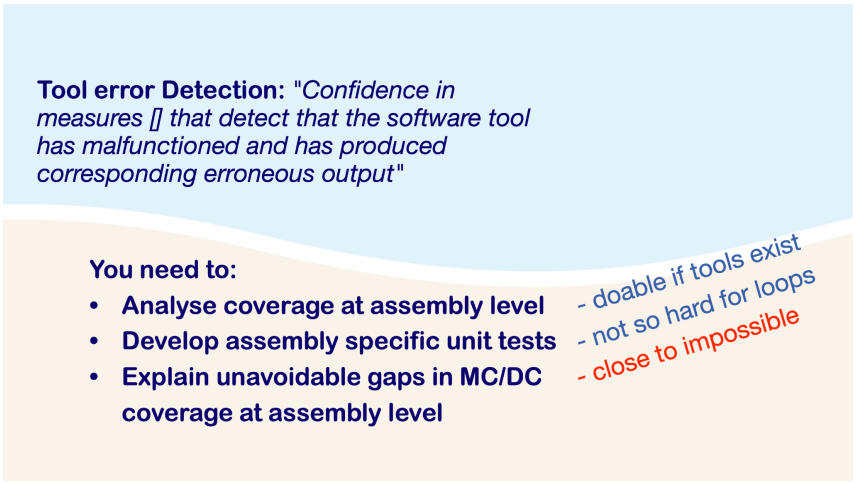

Figure 6: Ensuring confidence in the detection of compiler malfunctions

Back to the context of the analysis above. The original question was: can we skip compiler qualification by relying on application-level testing? The answer of ISO 26262 to that question is: yes, if you can *demonstrate sufficient confidence in the measures to detect that the compiler has malfunctioned and has produced erroneous output*.

But in our analysis of the assembly code for the simple loop it is shown that that is no simple task. To avoid untested code to be part of the application, you have to work hard. Not only do you need to have instrumentation in place for assembly-level code coverage analysis and generate additional tests to cover any gaps, but you must also be prepared for an in-depth analysis of redundant code that the compiler may have accidentally inserted in order to justify that it cannot be covered.

This requires a level of effort and expertise that goes beyond what is normally expected by ISO 26262 and it can be avoided by doing compiler qualification. After all, the goal of compiler qualification is to provide the confidence that it correctly translates the source code to target instructions so that the application developer does *not* have to worry about it.

Options for Open-Source Compiler Qualification

Now given that it is hard to rely on application-level testing to gain confidence in the compiler, ISO 26262 provides four options for compiler qualification. It does not differentiate between open-source compilers and other compilers, but here we will evaluate them from the perspective of an open-source compiler. The four qualification methods are:

- Increased confidence from use
- Evaluation of the compiler development process
- Validation of the compiler
- Development according to a safety standard

The ***increased confidence from use*** argument can be used if you already have so much experience with the compiler that you can confidently state that you know of any issues with it, because you have already used it for a sufficiently long time. The argument is hard to use because in practice compilers, and certainly open-source compilers, are regularly updated. Because of the complexity of compilers and their internal

architecture (a pipeline of stages where a defect affects every next stage), confidence in a previous version does not carry over to the next. Additionally, a change in the compilation options and even a different type of source code, stimulates a compiler in such a different way that it cannot be treated as the same compiler.

> The ***evaluation of the compiler development*** argument can be made if there is evidence that the development of the compiler is done in compliance to an appropriate standard such as Automotive SPICE. In practice, open-source compilers have guidelines for their development but they are never developed with functional safety in mind. They may originate from an academic or enthusiast project and, in the case of Gcc, span a development time of more than forty years. This makes it hard, if not impossible, to substantiate this argument.
>
> ***Validation of the compiler*** means to provide evidence that the compiler complies with its specification. This method treats the compiler as a black box and evaluates it as-is, without regard for its provenance. It can be applied to any compiler, including open source. The advantage of C and C++ compilers is that there is a well-defined standard that defines them, and that (commercial) test suites exist that can be used to verify them.
>
> ***Development according to a safety standard*** is the final option for compiler qualification but to the best of our knowledge, no compilers exist (open source or not) that are developed like this. It requires, for example, that their software development adheres to Part 6 of the ISO 26262 standard and can show evidence for it. No compilers exist that were created accordingly.

In short, the third method, qualification by validation of the compiler, is the most versatile method. It can be applied to any compiler. The tools and framework that works with one version of the compiler for a specific use case can be reused for a new version or a different use case. If the programming language is the same, it can even be used for a different compiler. This reduces the risk of compiler lock-in. Although that is not a safety risk per se, for products with a life span of decades it is a risk to consider.

A further advantage of compiler qualification is that it can be mostly decoupled from the application development time-line. “Mostly” because one needs to establish the use case for the compiler but as stated, it is not difficult to change the use case once the qualification framework is set up.

Validation of the Compiler

It is good to realize that at this point in the paper we are already well under way with the process that ISO 26262 defines for compiler safety. It starts by evaluating the compiler. We have done that by establishing that the compiler has a significant impact on the correctness of the generated code (which ends up in the hardware) and that without substantial effort, application-level testing cannot be relied on as a detector of defects in the compiler. Qualification by validation is also the most versatile option.

The next step in the process is validation of the compiler by verifying that it complies with its specification. This covers a couple of aspects:

- Verification of the compiler. For this we need a specification of the compiler to verify against. For C and C++, and a few other programming languages this is easy because there are well-established, internationally approved, standards. This is not the case for many other programming languages. While they are all properly documented, documentation is often unclear when it comes to edge cases of behavior. An advantage of programming language standards is that they describe the behavior of the program at execution time. They do not describe exactly what assembly instructions must be

generated. This leaves freedom to the compiler to optimize code and it makes it possible that one language standard is valid for many different types of implementations. A further advantage is that it allows us to include the whole toolchain (compiler, assembler, linker, loader) to be considered part of the “compiler” because they are all included in the execution framework that establishes the required behavior of the program. It does not mean that all of these components are qualified as stand-alone components. It means that the toolchain as a whole is capable of implementing the programming language.

Next, we need a test suite that is organized in such a way that it defines the relation between the language specification and the tests. This is a non-trivial requirement because most test-suites, including those maintained for the common open-source compilers, are built and maintained as regressions suites - they are a catalogue of tests for bug fixes. Fortunately, for C and C++ commercial offerings exist that are properly organized.

And finally, one needs to establish the use case of the compiler. This is mostly related to the compiler options used by the application, but it can include other aspects of the environment. It must include all variables that can potentially change the generated instructions.
With these ingredients, we can run the test suite for the given use case and have confidence that all aspects of the language specification are verified.

- The next step is to verify how the implementation reacts to incorrect input. The C and C++ standards define that the compiler must generate diagnostics for syntax and typing violations. Therefore, these checks should be included in the test suite. They do not require, and this is not expected, that diagnostics are generated for programs with undefined behavior, not even at run-time. This is a weakness of these programming languages that also has positive aspects, such as efficiency, and which must be documented in the compiler safety manual that results from the qualification effort. It can be mitigated by the use of programming guidelines and static analysis tools.
- The final step is the analysis of the results of testing. It is not necessary to fix any defects. Although the open-source community can be extremely quick to provide a fix (the author once received a patch to Gcc within hours of the report, on a Sunday morning), there is no expectation of any timeline for repair. The objective is not to prove the compiler is flawless, as no existing compiler is perfect. Rather, the focus is on documenting identified defects and providing clear workarounds. This process ensures that application developers are aware of potential issues and can mitigate them. By addressing these typically specific and avoidable behaviors, we systematically build confidence in the compiler as a dependable tool for safety-critical development.

The final requirement of ISO 26262 is documenting the complete qualification process (not only the test results, the work arounds, but also the quality of the test suite itself as used in the process as well as the confidence therein). Although not required as such, it is often useful to create a separate safety manual that is focused on the application developer. It contains a description of the limits of the use case for which the qualification is valid, and the list of defects and mitigations that they can apply in the application development process.

And What About Testing Optimizations?

Even for a dedicated, requirements-based, test suite for a language implementation it is not a given that it properly implements optimization testing. The reason is that in the compiler, the implementation of

optimizations comprises a significant part of the compiler source code. But in the language specification it occupies not more than a single paragraph that states that an optimization shall not change the behavior or the program. Since this is a negative requirement (it requires the absence of change), testing it exhaustively requires an infinite number of tests, which is neither practical nor actionable.

Optimization testing often relies on running benchmarks because they contain code that is the target of optimizations. Although that may say something about the efficacy of the optimization, it seldom provides a good test. This is because benchmarks, like application-level tests described earlier, are also not targeting the special cases of an optimization. Furthermore, many benchmarks do not verify the result of the test which means that a defect will not be diagnosed.

In a good test suite (i.e., SuperTest from Solid Sands), you should expect a substantial collection of optimization tests showing the following properties. The first is that test tests exist to systematically trigger every optimization. In many compilers, individual optimization stages can be turned on and off. By verifying that this changes the generated code, we know that the test interacts with the optimization. This helped us find gaps in optimization coverage for which we then created new tests. The second is that the tests have sufficient assembly level code coverage. For SuperTest, we created additional tests and test cases until we verified that assembly code coverage is maximal.

A Safety Certified Compiler

The documents produced for the qualification can now be taken to an assessor for independent review. This does not have to be an outside consultant, it depends on the level of independence required. ISO 26262 provides guidance for this. The most independent review is to get the process certified. Technically, certification does not mean that the compiler itself is guaranteed to be perfect or even certified. The task of the certification authority is to assess that the qualification process has been implemented faithfully. This is what is certified. When people speak of a “safety certified compiler,” it means “a compiler that has gone through a qualification process based on a specific functional safety standard for which the process and documentation where certified by an independent certification body and you should really read the documentation in order to understand what the range of safety related applications is for which this compiler can be used.”

Qualification of Open-Source Compilers

As we have seen above, there are no barriers to the qualification of open-source compilers for the ISO 26262 standard, and by extension for the whole family of IEC 61508 related standards. Referring back to Figure 1, this is different for the aviation standards. They do not allow the compiler to be treated as a black box and they require the documentation of the compiler’s development process and its internals. These objectives are difficult to achieve for open-source compilers. However, that this is not unique to open source, it is also true for proprietary compilers - possibly even to a higher degree because of their closed nature.

As an aside, compilers are used in developments for aviation but if they are not qualified it requires that their output, the generated assembly code, is inspected to verify that it matches the source code.

Conclusion

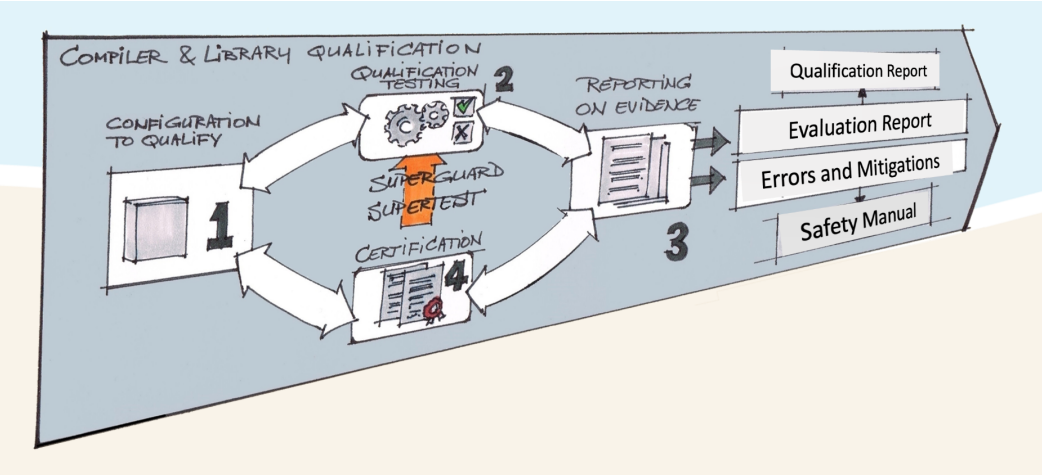

Figure 6: Compiler qualification is a repeatable process driven by standards such as ISO 26262

Awareness of software safety has grown a lot over the past fifteen years and the need for this is accelerating. A major driver is the transformation of cars into software defined vehicles.
Compilers are great tools that take away much of the complexity for application developers allowing application-level programming to be focused on higher levels of abstraction (C or C++ vs assembly). Compilers are non-trivial tools with considerable internal complexity, often more complex than the application that is being compiled. For that reason, they cannot be ignored in the process of qualifying an application for use in a safety-critical component. Compiler qualification falls under (off-line) tool qualification processes defined by IEC 61508-like functional safety standards. These processes are less strict than general application development processes because the tool does not become part of the safety-critical component.

Verification of the compiler can be used as the method to qualify it. This method permits treating the compiler as a black box. Given that language specifications define the behavior of the implementation, which includes the toolchain, this also makes it possible to consider the assembler and linker as part of the black box.

This is not true for libraries. Although standard libraries for C and C++ are defined by the same language standards, libraries cannot be considered part of the compiler tool. Libraries are compiled by the compiler and their code is linked with the application and becomes part of the safety-critical component. For libraries, the software related processes described in Part 6 of ISO 26262 must be applied, similar to application software in general.

An important advantage of compiler qualification is that it provides confidence that the compiler does its job. This means that compiler verification can be decoupled from the verification of the other parts of the safety-critical component - verification of the software can stop at the source code level and does not have to consider defects in the compiler beyond those described in the compiler safety manual. This benefit can extend to additional projects if the same compiler is used there, saving time and effort.