WIND RIVER | VALIDAS

# ISO 26262–Compliant Usage of Wind River Diab Compiler

*Graham Morphew of Wind River and Dr. Uwe Müller, Dr. David Seider, and Dr. Oscar Slotosch of Validas AG*

## Table of Contents



### Executive Summary

The draft safety standard ISO 26262 has a new approach to ensuring that the tools used to develop software for automotive application do not affect the safety of the developed system. The key idea is to ensure that all potential errors related to the tools can be avoided or detected. To do this, ISO 26262 requires that developers determine a tool confidence level (TCL) consistent with the way the tool is applied. Depending on the desired Automotive Security Integrity Level (ASIL), ISO 26262 recommends qualification methods for a given TCL. The highest TCL (TCL3) requires an intensive qualification method (e.g., a validation suite), while TCL1 requires no qualification method for the tool. However, achieving a low TCL requires that you be able to detect potential errors in the process of using the tool, which in turn will involve more effort in integrating error checks and restrictions into the development process.

This paper provides information to users of Wind River Diab Compiler about tool qualification according to ISO 26262 and shows how TCL can be determined and reduced. This paper proposes a qualification method for Wind River Diab Compiler that combines validation and the application of a safety standard. The method is based on the idea of a qualification kit for the compiler that can be applied to justify a high TCL without requiring manual steps for error detection or other tools.

### Tool Qualification in ISO 26262

ISO 26262 is an adaptation of IEC 61508 that addresses the specific needs of automotive electric/electronic systems used in mass-produced passenger road vehicles with a maximum weight of 3.5 tons.

ISO 26262 is based on the idea of safety goals. These are elaborated for each automobile system, for example, the airbag system. (ISO 26262 distinguishes between "elements," "items," and "systems." This paper groups all of these under the term "system.") The analysis starts with a hazard and risk assessment from which a safety goal is derived. In the example of the airbag system, a hazard would be unintended deployment; the safety goal would be to ensure the airbag does not deploy unless a collision happens.

The corresponding ASIL is determined for the airbag system, with level A representing the least stringent safety measure for avoiding an unreasonable residual risk and level D the most stringent. Next, a functional safety concept states how to achieve the safety goal, such as introducing a redundant function for detecting a collision. Then a technical safety concept addresses the actual implementation in terms of hardware and software. For hardware, it might propose using two orthogonal acceleration sensors, for example; and for software, it might propose that the two sensor input data streams be treated as exchangeable.

During the development of software for a safety-critical system, it is quite common to use tools. ISO 26262, part 8, section 11 gives guidance on the qualification of software tools. One goal of this paper is to provide an understanding of the tool qualification concept established in ISO 26262 and explain how qualification of Wind River Diab Compiler for use in an ISO 26262 project can be achieved by using a novel approach to determine the TCL of a specific use case of the compiler, exploiting the data flow character of a toolchain for the purpose of tool qualification.

In the context of ISO 26262, a software tool simplifies or automates activities and tasks required for the development of a safety-related system. (Note that ISO 26262 requires that all tools be considered independent whether they are in-house, freeware, or commercial.) The overall objective of tool qualification is to provide evidence that the tool is suitable to develop a safety-related system in such a way that one can ensure confidence in the correct execution of the tool.

To establish that confidence, ISO 26262 asks two questions and derives a well-defined confidence level from the answers. The first question deals with tool impact: Does a malfunctioning software tool and its erroneous output lead to the violation of any safety requirement allocated to the safety-related system being developed? If the answer is no, the tool is classified as TI0 (zero tool impact). If the answer is yes, the second question asks about the probability of detecting or preventing tool errors in the output of the software tool. Probability in the context of tool qualification denotes a qualitative measure not a concrete number.

Before determining the tool confidence level, ISO 26262, part 8, section 11 requires that you collect the following information for every tool in use:

- Identification and version number of the software tool
- Configuration of the software tool
- Use cases of the software tool
- Expected environment of the tool
- Behavior under anomalous operating conditions
- Known software malfunctions
- Measures for the detection of malfunctions

Next, tool impact (TI) is clarified. ISO 26262 distinguishes two classes of tool impact:

- TI0 indicates there is no possibility that the tool introduces or fails to detect safety-impacting errors in the developed software.
- TI1 applies to all other cases.

In the next step, you elaborate the probability of detecting or preventing tool errors in the output of each tool. ISO 26262 divides the probability into four tool detection (TD) levels:

- TD1 indicates a *high* degree of confidence that a malfunction or an erroneous output from the software tool will be prevented or detected.
- TD2 indicates a *medium* degree of confidence that a malfunction or an erroneous output from the software tool will be prevented or detected.
- TD3 indicates a *low* degree of confidence that a malfunction or an erroneous output from the software tool will be prevented or detected.
- TD4 applies to all other cases.

Finally, the TCL is derived from the TI and TD, as summarized in Table 1.

**Table 1: Calculation of TCL by TI and TD, According to ISO 26262, Part 8, Section 11**

| | TD1 | TD2 | TD3 | TD4 |
|---|---|---|---|---|
| TI0 | TCL1 | TCL1 | TCL1 | TCL1 |
| TI1 | TCL1 | TCL2 | TCL3 | TCL4 |

Note: TD4 will likely be eliminated in the final standard, leaving three levels. In this case, the TCL determination would be made according to Table 2.

**Table 2: Calculation of TCL if TD4 Is Eliminated**

| | TD1 | TD2 | TD3 |
|---|---|---|---|
| TI0 | TCL1 | TCL1 | TCL1 |
| TI1 | TCL1 | TCL2 | TCL3 |

Once the TCL is determined, a qualification method is chosen. In contrast to the TCL calculation, the qualification methods are not expected to change in the final standard. The qualification method is derived from two inputs: the ASIL of the safety-related system (documented in ISO 26262 "safety plan") and the determined TCL.

There are four different qualification methods proposed in the standard:

- Increased confidence from use
- Evaluation of the tool development process
- Validation of the software tool
- Development in compliance with a safety standard

The standard also provides individual chapters for each qualification method. These chapters include a definition of terms, criteria, and instructions needed for a tool qualification (see ISO 26262, part 8, sections 11.4.5–11.4.8).

A tool rated TCL1 does not need any further qualification independent of the ASIL because it has either no influence of the safety functions or its potential errors are detected with a high probability. For all other TCLs (and depending on the ASIL), the software tool must be qualified according to the methods shown in Tables 3, 4, and 5. For a given ASIL, one qualification method marked with "++" (highly recommended) must be applied. Methods highly recommended for higher ASILs can also be used for lower ASILs.

Table 3: Qualification Methods for a Software Tool Classified TCL4, According to ISO 26262, Part 8, Paragraph 11.4.4.2

| Methods | | ASIL | | | |
|---|---|---|---|---|---|
| | | A | B | C | D |
| 1a | Increased confidence from use | ++ | ++ | + | o |
| 1b | Evaluation of the tool development process | ++ | ++ | ++ | + |
| 1c | Validation of the software tool | + | + | ++ | ++ |
| 1d | Development in compliance with a safety standard | + | + | ++ | ++ |

Table 4: Qualification Methods of a Software Tool Classified TCL3, According to ISO 26262, Part 8, Paragraph 11.4.4.2

| Methods | | ASIL | | | |
|---|---|---|---|---|---|
| | | A | B | C | D |
| 1a | Increased confidence from use | ++ | ++ | ++ | + |
| 1b | Evaluation of the tool development process | ++ | ++ | ++ | ++ |
| 1c | Validation of the software tool | + | + | + | ++ |
| 1d | Development in compliance with a safety standard | + | + | + | ++ |

Table 5: Qualification Methods of a Software Tool Classified TCL2, According to ISO 26262, Part 8, Paragraph 11.4.4.2

| Methods | | ASIL | | | |
|---|---|---|---|---|---|
| | | A | B | C | D |
| 1a | Increased confidence from use | ++ | ++ | ++ | ++ |
| 1b | Evaluation of the tool development process | ++ | ++ | ++ | ++ |
| 1c | Validation of the software tool | + | + | + | + |
| 1d | Development in compliance with a safety standard | + | + | + | + |

As stated previously, the final standard is likely to have three tool confidence levels: TCL1, TCL2, and TCL3. In this case, the qualification methods would be chosen according to the scheme shown in Tables 6 and 7.

Table 6: Qualification Methods for a Software Tool Classified as TCL2, According to the Expected Final Standard

| Methods | | ASIL | | | |
|---|---|---|---|---|---|
| | | A | B | C | D |
| 1a | Increased confidence from use | ++ | ++ | ++ | + |
| 1b | Evaluation of the tool development process | ++ | ++ | ++ | + |
| 1c | Validation of the software tool | + | + | + | ++ |
| 1d | Development in compliance with a safety standard | + | + | + | ++ |

Table 7: Qualification Methods of a Software Tool Classified TCL3, According to the Expected Final Standard

| Methods | | ASIL | | | |
|---|---|---|---|---|---|
| | | A | B | C | D |
| 1a | Increased confidence from use | ++ | ++ | + | + |
| 1b | Evaluation of the tool development process | ++ | ++ | + | + |
| 1c | Validation of the software tool | + | + | ++ | ++ |
| 1d | Development in compliance with a safety standard | + | + | ++ | ++ |

All methods have drawbacks and are difficult to apply for the validation of a tool:

- **Confidence from use:** This is very difficult to achieve because sufficient data is usually not available for the tools containing the required information, both for the concrete tool version and the use cases. In addition, a systematic collection of usage data, environments, and applications is usually not available. Furthermore, it is not clear whether all errors are reported to the tool supplier because this is often an additional burden for users.
- **Evaluation of the tool development process:** This requires that the tool be developed according to a defined process, satisfying national or international standards. This is not the case for most of the tools currently available, even if there are prominent exceptions known.
- **Validation of the software tool:** This shows that the tool satisfies its requirements using systematic tests. Also, reaction to abnormal usage conditions has to be tested. Depending on the complexity of the tool, this can be quite expensive.
- **Development according to a safety standard:** This is difficult because safety standards usually are not established for tool development and require adaptations.

Because every qualification method has individual drawbacks, it is best to reduce the TCL so that no qualification method needs to be applied. However, this can only be done by extending the development process with additional checks to discover or prevent the potential errors. In some cases, these extensions can be done using extra checking tools, such as a MISRA C code checker or reference tools. In other cases, the extension to reduce the TCL requires manual steps such as reviews. In both cases, the process extensions require additional efforts and costs that could be saved by using sufficiently qualified tools.

The value of a qualified tool with a high TCL depends on the costs of the alternatives in the process and the frequency the process is applied. For example, if a tool is applied every day and a review of its output is required to reduce the TCL to TCL1, the review costs could be quite high; tool qualification is much cheaper, even under the previously mentioned restrictions of the applicable tool qualification methods.

This paper proposes a qualification method for Wind River Diab Compiler that combines validation and the application of a safety standard. The method, used by Wind River partner Validas AG, a software engineering firm in the field of safety-based systems that require certification for ISO 26262, IEC 61508, DO178B, and AUTOSAR, is based on the idea of a qualification kit for the compiler that can justify a high TCL and not require manual steps or other tools. This qualification method should be applied if the effort related to error detection for each application of the tool in a given period is higher than the costs of the required tool qualification. The costs of tool qualification, which are constant for a given version, pay off during the repeated application of the qualified tool. Every application of a qualified tool saves costs compared with a nonqualified tool.

The ISO 26262 tool qualification process is summarized as follows:

- The user determines the tool version, configuration, and used features.
- The tool supplier provides information:
  - Known bugs
  - Behavior in abnormal operating conditions
  - Detection possibilities of malfunctions
- The user determines the TCL based on the planned application process of the tool.
- If the tool or toolchain is assessed with TCL1, no tool qualification is required. If the tool or toolchain is assessed with a higher TCL, the tool supplier supports tool qualification by providing a qualification kit. (Tools that have a qualification kit are called “qualifiable” or “certifiable.”)
- The user qualifies the tool.

When approaching tool qualification according to ISO 26262, the first consideration with respect to the TCL is the compiler and the linker, but supporting tools such as the make process also need to be evaluated to ensure they will not introduce new errors (e.g., a dependency is overlooked and the software is not made correctly).

Table 8 is an example of a partial list of errors and checks that would impact TI and TD assessment. It is a based on the process of performing a compilation and linking of a given piece of C code using a Makefile-based build process. The example aims to illustrate the concept of evaluating the error risk related to TI and the checks that help detect the errors (related to TD) and does not intend to show all possible existing errors in such a toolchain.

Table 8: List of Assumed Errors and Checks

| Item | Tool | Error/Check (Probability) | Error Description |
|---|---|---|---|
| 1 | Compiler | Error | Truncates long function names |
| 2 | Make | Error | Misses dependency |
| 3 | Compiler | Check (high) | Finds syntax errors in C code |
| 4 | Linker | Check (medium) | Finds truncated function names |

### Toolchain Analysis

Toolchain analysis is a method developed by Validas—and not explicitly prescribed by ISO 26262—to build a simple toolchain model that checks for correct detection of potential tool errors by exploiting the toolchain itself. This section describes the methodology and demonstrates how ISO 26262 requirements for TCL determination are fulfilled. Validas and Wind River have applied this methodology in various automotive projects and gained useful process improvements.

An example of toolchain analysis is when a linker can detect compiler errors but the compiler cannot detect linker errors because the data flow is from the compiler to the linker, via the object file that is created by the compiler and read by the linker. There is no flow from the linker to the compiler.

For a given tool, the list of errors is identified by reviewing the available tool defect list or by conducting an expert assessment. In addition, it needs to be determined which errors of other tools can be detected or avoided by the tool under consideration. (This kind of work is in keeping with ISO 26262, part 8, section 11.4.2.)

### Simple Toolchain Model

The analysis of the toolchain can be modeled by creating a diagram of the topology of the process flow in the toolchain using arrow-shaped boxes to denote the tools or tool use cases. You can then connect the arrow-shaped boxes using solid lines to denote data exchanges or dotted lines if one tool is called by another tool. Once this basic graphical model is created, you can add the known errors for each tool represented in the model using a lightning bolt symbol. To complete the model, attach checks and restrictions to each individual tool. A stop sign symbol can be used to denote these checks and restrictions, which represent the way the tool will find or avoid a certain error. The terms "check" and "restriction" are used because they are very intuitive and denote that the tools are connected by data or a call in the toolchain:

- A check can only detect errors that occur earlier in the chain.
- A restriction can only avoid errors that occur later in the chain.

The actual analysis starts by connecting the errors of a certain tool with the checks and restrictions of the other tools in the chain. This is done using a dotted line with an arrowhead, indicating the direction of the process flow.

Here is a simple set of terms that describe the relevant aspects of tool connections within a toolchain. This is the starting point for consistency checks and determination of the TCL:

- **Artifact:** This term usually denotes data exchanged between use cases of tools.
- **Tool:** Tools are subdivided into separate "use cases." A compiler used only with a specific option set is a tool with a single use case. "Tool" can also refer to human activities.
- **Use case:** This can input and output artifacts and "call" other use cases of tools, referred to as "control flow."

Tools and artifacts from the example in Table 8 are modeled in Figure 1. The boxes represent tools or tool functions, while the lines connect the tools and build up the "flow." Solid lines describe the data flow by artifacts handed over between tools, and the dotted lines illustrate the control flow, for example, the make tool calling compiler and linker while processing a Makefile.

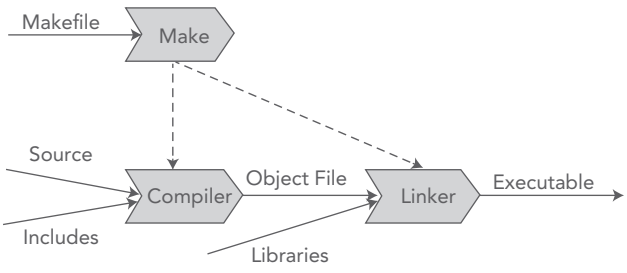


*Figure 1: Simple toolchain model based on make*

For the TCL determination, there are additional useful terms and notations:

- Potential "errors" are assigned to tool use cases.
- "Checks" and "restrictions" of errors (with a probability of low, medium, or high) are assigned to individual use cases.
- Dotted lines are used to model the error detection or prevention between different tools.

The lightning bolt symbol (as shown in Figure 2) is for modeling errors, and the stop symbol is for modeling checks and restrictions of errors. The stop symbol indicates the tool error detection probability of ISO 26262:

- H means high probability.
- M means medium probability.
- L means low probability.

Figure 2 incorporates the errors listed in Table 8. The bold dotted line indicates that compiler error No. 1 ("Truncates long function names") is detected by linker check No. 4 ("Find missing function names").

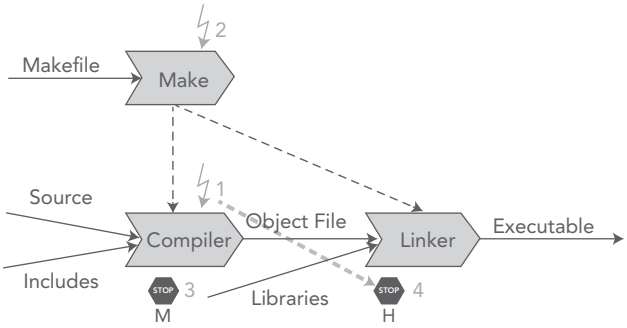


*Figure 2: Enriched model for make with errors, checks, and detection link*

The next step is to calculate the TCL as described previously:

- The linker has no potential errors, so no impact, and is classified TCL1.
- The compiler has only one potential error that would be detected with medium probability and therefore is classified TCL2.
- The make tool has one undetected potential error and is therefore classified TCL3.

The result of the analysis is illustrated in Figure 3, with tool confidence levels TCL3, TCL2, and TCL1 represented.

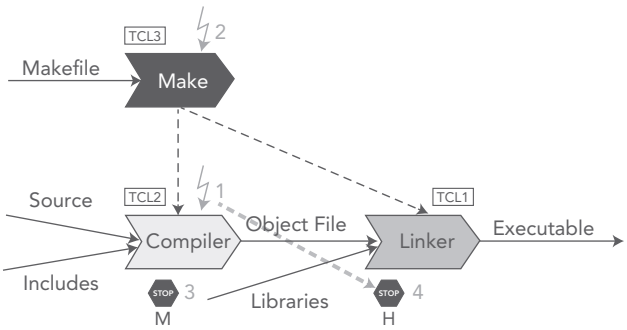


*Figure 3: Analyzed model for make-determined TCLs*

### Reducing the Criticality of the Make Tool

In this section, the process for building software is modified to reduce the criticality of the make tool. To do this, the option "–n" after a "make clean" is used. This produces a log file as output that contains all commands that will be executed. This log file is reviewed against the Makefile so that missed dependencies are detected with a high probability. This review can be improved by storing the last log file and comparing it using diff. This can be done automatically in all cases where the Makefile is not changed. The resulting model is depicted in Figure 4, with a newly introduced review step in the upper right side.

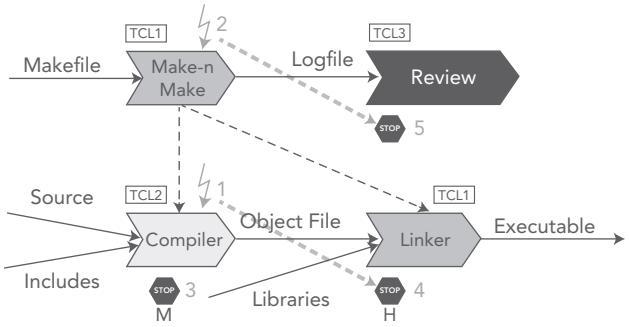


*Figure 4: Flow with enhanced confidence reached by the activities and data shown in upper right*

Table 9 demonstrates the tool error analysis for the modified toolchain. In both representations, the make utility has become TCL1 because of the addition of the review of the log file.

Table 9: Tool Error Analysis and TD Calculation for the Model Shown in Figure 4

| Item | Tool | Error/ Check (Prob-ability) | Error Descrip-tion | Item Detect-ing Error | Tool Error Detec-tion |
|---|---|---|---|---|---|
| 1 | Compiler | Error | Cuts long function names | 4 | TD 2 |
| 2 | Make | Error | Misses depen-dency | 5 | TD 1 |
| 3 | Compiler | Check (high) | Finds syntax errors in C code | | |
| 4 | Linker | Check (medium) | Finds truncated function names | | |
| 5 | Review | Check (high) | Finds over-looked depen-dencies | | |

### Comparison of the Toolchain Models

The make tool is used for building software in two different processes with two different models. The first variant (Figure 3) ends up as a TCL3 for the make tool, while the improved variant (Figure 4) delivers TCL1 for make. This shows that the TCL of a tool does not depend on the tool or its potential errors but on the process in which it is applied.

In principle, it is possible to extend the process so that every used tool has TCL1. However, the extended processes require additional steps to detect the errors. In the make example, the review is a partially manual step. Depending on the qualification costs of the make tool and the frequency of the application of the make tool, the user can decide whether to qualify the tool and to apply the simplified process with TCL3 or to use an unqualified make tool with TCL1 within an extended process in which errors of the tool are detected.

## Analysis of Wind River Diab Compiler

As stated previously, it must be ensured that tools used for developing safety-critical applications do not violate safety requirements. The measures that must be taken and the methods used depend on the TCL and ASIL.

A compiler is one of the core tools to generate embedded software so it has an enormous impact on all safety-critical parts related to the software. Consequently, a compiler must be analyzed and, depending on the TCL, even qualified when used in a project covered by ISO 26262.

First, based on an analysis of the tool, the TCL must be determined. This section outlines the required information and demonstrates how such an analysis could be performed.

This paper does not focus on any specific version or configuration of Wind River Diab Compiler or any particular analysis done by Validas. It does not present an analysis of any known defects in Diab Compiler. This analysis needs to be done for each customer configuration individually by using defect tracking in the Wind River Online Support system, which provides a selection of relevant known issues for the tool.

Wind River Diab Compiler provides many features and supports many use cases, such as compilation with different optimization switches, different target platforms, and just linking. In regard to safety analysis, this paper focuses on the main functionality of generating an executable binary. Other use cases are not addressed in detail here.

ISO 26262 requires an evaluation of potential threats (violations of safety requirements due to tool malfunctions). Such an analysis has to be done for each customer project. Since this requires extra work, it makes sense to prepare a general concept of potential tool malfunctions and measures to detect or avoid them.

To show how a compiler can be analyzed, similar problems are grouped into one class if they can be detected or prevented in the same way. For an analysis of a concrete use case this classification will have to be adapted appropriately.

### Expected Environment

The main elements of the compiler environment are the host on which the complier is installed and, depending on the setup, the servers the host is connected to.

The Wind River Diab Compiler requirements for the host computer are stated explicitly in the release notes and must be considered. For example, on the Microsoft Windows Vista Enterprise x86-32 operating system, Wind River Diab Compiler version 5.8.x requirements include the following:

- **RAM memory:** 1GB (2GB for larger projects)
- **Disk memory:** 500MB for installing all supported architectures
- **CPU:** Intel Pentium 4 class computer with 2GHz processor or higher

This assures that enough system resources are available during operation. If the tool is not used locally, the host computer needs to have a network connection to contact the license server.

### Behavior Under Anomalous Operating Conditions

Depending on the use case, many anomalous operating conditions can be relevant. This section provides some examples and potential behaviors of the compiler.

If the environment requirements are not fulfilled, the compiler is expected to abort with an error message such as "Wrong Operating System," "Missing Disk Space," or "No License." It might also significantly slow down the CPU and memory access so that the compilation seems not to terminate. In this case, even the code generation could be affected, producing incomplete code.

Another anomalous operating condition might be an incorrect tool installation such as wrong configuration, incomplete installation, wrong settings of environment variables, or conflicts due to different versions installed. In such cases, the expected behavior is not predictable and the tool might react in various ways:

- It might not work at all, and with no notification.
- It might abort with an error message.
- It might run smoothly and deliver output that is correct by chance or erroneous.

Finally, the user could run the tool with a prohibited combination of switches or features. In this case, the compiler should abort the compilation and clearly report the error.

### Description of Possible Malfunction Classes

The only tool malfunctions ISO 26262 considers relevant are those that can violate a safety requirement. For a compiler, a safety requirement is violated if the final executable is affected by the tool error. This section considers these kinds of possible errors. Countermeasures to help detect these kinds of errors are described earlier with the make utility example. These are principal classes of potential errors, as covered by ISO 26262, and are not based on known compiler bugs, which have to be considered separately for each selected version and use case.

#### *Incomplete or Encumbered Generation of Executable Code*

The examples of possible anomalous operating conditions give rise to the following potential malfunctions:

- **PE001:** No detection of insufficient resource availability (RAM, CPU, etc.) on the PC during compilation by the compiler, resulting in generation of erroneous code
- **PE002:** No recognition of a corrupt installation of the compiler, resulting in generation of erroneous code
- **PE003:** Compilation with a prohibited or contradictory combination of switches and features, resulting in generation of erroneous code
- **PE100:** Generation of an incomplete executable due to abort, which happens if the compiler exits or crashes during the build process due to an internal error; the compiler would stop compilation with an incomplete executable and return an error code

#### *Generation of Erroneous Codes*

The compiler could generate erroneous codes without producing an error message or warning, due to an internal compiler error. The following are some of the possible error classes:

- **PE200:** No recognition of lexically, syntactically, or semantically erroneous code. The compiler does not recognize that the source code is lexically, syntactically, or semantically incorrect (in the sense that it is not compliant to the C standard). In all of these cases, the behavior of the resulting executable code is not predictable and can lead to arbitrary results.
- **PE201:** Non-runnable code. The object code ends up in a binary that cannot be run or crashes during execution. Causes include wrong ABI, wrong linking information, or a binary that is too large for the target.
- **PE202:** Problems with type casting information. The compiler generates code where implicit or explicit type casts are realized in the wrong way. This can especially affect values near the boundaries of the ranges of data types.
- **PE203:** Problems with floating-point arithmetic. Numerical deviations occur due to errors in realization of floating-point calculations in generated code.

- **PE204:** Problems with integer arithmetic. Large numerical deviations can occur due to errors in realization of integer calculations in generated code.
- **PE205:** Problems with conditional or unconditional jumps. The compiler generates code in which jump conditions are not determined correctly or jumps are not realized as intended.
- **PE206:** Problems with pointer arithmetic. The compiler generates code where the handling of pointers is not realized correctly.
- **PE207:** Problems with general optimization. The compiler generates code where optimization is realized incorrectly. Examples include the following:
  - Wrong modification of numerical and logical expressions
  - Wrong recognition and realization of reachability in source code
  - Errors with range checks and namespaces
  - Errors in constant folding
  - Wrong allocation of registers
- **PE208:** Problem with target specifics. The compiler generates erroneous code due to tailoring to the target platforms (e.g., a certain microcontroller). The following errors could occur:
  - Errors in FPU emulation or invocation
  - Code with wrong access to the target hardware
  - Data type errors (e.g., target only supports 16-bit data types, which is not observed correctly in the executable code)
  - Endianness realized wrongly in the executable

### Measures for Detecting Potential Errors

The following are ways of detecting the potential errors mentioned in the previous section (with detection probabilities in parentheses):

- **PE100 (high):** Aborts can be detected by checking the compiler's return code. This is usually done within Makefiles. If other command files are used for compilation, you must ensure that they detect the error codes from the compiler.
- **PE002 (medium), PE201 (high):** A simple target test can be performed. The binary is loaded to the target and executed there. If it is not runnable, this will be detected.
- **PE208 (medium):** Errors related to hardware-specific parts such as access to memory-mapped I/O or register access can be detected by running a hardware integration test. These kinds of tests cannot be executed on the host.
- **PE202, PE204, PE205, PE206, and PE207 (high if combined with structural tests; otherwise low), PE203 (medium):** Back-to-back tests on the host system with a reference compiler can show the correct implementation of hardware-independent parts. In floating-point operations, numerical effects can lead to small deviations due to differently implemented floating-point arithmetic on host and target. These deviations can add up to larger deviations in more complex calculations. Determining whether larger floating-point deviations are due to numerical effects or errors in the code requires a manual analysis. Small floating-point deviations that originate from a malfunction in the compiler cannot be detected with a high probability using this method; for detection of such errors, back-to-back tests on the target have to be run. In a model-based approach, a back-to-back test can be performed against simulation results of the model from which the source code is generated.
- **PE202, PE204, PE205, PE206, and PE207 (high if combined with back-to-back tests; otherwise low), PE203 (medium):** A structural test of the code should execute the binary and demonstrate that all parts of the code have been executed. Since instrumentation of the target code is more difficult than instrumentation on the host, the structural test can be split into two tests: one on the host, showing that the required coverage has been reached, and one on the target, showing that the code is working. Since hardware-dependent code cannot be executed on the host, it has to be stubbed on the host.
- **PE203 and PE208 (high):** Errors in floating-point calculations or wrongly realized target specifics can be detected by back-to-back tests on the target system using a reference compiler generating code for the target.
- **PE200 (medium):** Running the compiler with an option that switches on all warnings (e.g., –Xlint for Wind River Diab Compiler or –Wall for a GNU compiler) and reviewing the log files can give hints on possible syntactical, semantic, or logical errors in the source code.
- **PE207 (high):** Errors in optimization can be detected by using back-to-back tests with different optimization levels of the compiler being analyzed.

### Measures for Avoiding Potential Errors

The following describes ways to avoid potential errors described earlier in the section "Description of Possible Malfunction Classes," (with probability of avoidance in parentheses):

- **PE001 (high):** Potential resource errors can be avoided by applying manual checks of memory and CPU usage before and during the compilation, for example, by using the Task Manager in Microsoft Windows or a tool such as ps on Linux/UNIX. The application guide has to communicate clearly to the user that during compilation no other activities should be started on the computer.
- **PE002 and PE003 (high):** These errors can be avoided by performing manual reviews. To check whether the installation is corrupt, review the contents of the installation folders and the log files of the installation procedure. To avoid a wrong combination of compiler options or features, perform a crosscheck with the user manual of the tool or with the tool supplier.
- **PE200 (high):** These errors can be avoided by applying additional code checkers (e.g., MISRA C) that check compliance to certain standards as well as check for other errors.

- **PE206 (high):** These errors can be avoided by a static analysis for allocation of memory in the code. Restricting yourself to a safer subset of the programming language (such as forbidding the use of pointer arithmetic or ensuring compliance with the MISRA C coding standard) avoids the possibility of this kind of error.
- **PE204 (high):** Possible compiler errors in conjunction with floating-point calculations on the target can be avoided by forbidding the use of floating-point data types in the source code.
- **PE207 (high):** These errors can be avoided by disabling optimization while building the executable.

### Assessment of Coverage-Based Qualification

As explained in the earlier section "Tool Qualification," evidence of suitability must be provided for tools used to develop safety-related applications. To do this, a TCL is determined and, depending on the TCL and the ASIL levels, certain methods for qualification are applied. This section explains why all qualification methods listed by ISO 26262 are very costly. The method that turns out to be the most feasible—up to the highest TCL and ASIL levels—is the validation of the software tool.

This section describes how to develop a lean process for validating a compiler. This process is based on a qualification test suite and an efficient metric for determining that all relevant parts of the compiler have been covered by the test suite. Figure 5 gives an overview of the possible task sharing: The compiler supplier delivers a compiler qualification kit (including a test suite, appropriate tools, user manuals, process description, etc.), and the customer performs the validation of the tool in the environment for the relevant use cases, possibly with support by the vendor.

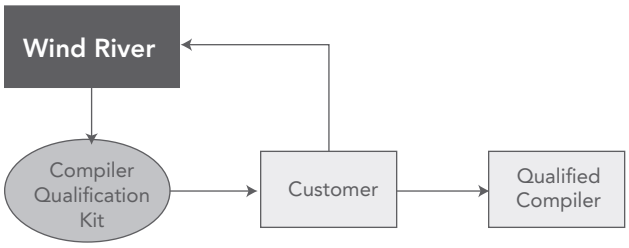


*Figure 5: Basic validation process*

ISO 26262 requires that the following criteria be satisfied for the validation of a software tool (ISO 26262, part 8, section 11.4.7):

- Validation measures shall demonstrate that the software tool fulfills its specified requirements within the determined coverage of those requirements.
- Malfunctions or erroneous outputs of the software tool occurring during validation shall be analyzed together with information on their possible consequences and with measures to avoid or detect them.
- Reaction of the software tool to anomalous operating conditions shall be examined.
- Robustness of the software tool shall be examined (i.e., in conditions with excessive or complex input).

All of these criteria can be satisfied by the application of an appropriate qualification test suite with suitable test cases. However, the core task in validating a tool is providing evidence that all relevant requirements of the customer are fulfilled (see the first criterion in the previous list). To satisfy this criterion, the approach described in the next section is suggested, based on a source code coverage analysis of the compiler.

### Validation Based on Source Code Coverage

Figure 6 describes the basic concept for measuring whether all customer requirements for a compiler are covered. The largest box comprises the whole source code base of the compiler, which is itself made up of functions, global variable definitions, and so on. Let's assume that the compiler has been extensively tested by test suites A and B and that all results have been analyzed thoroughly. The compiler is considered to be validated for the part of the source code covered by the test suites (in this case, function 1 is completely trustworthy). If a customer uses the compiler for a certain project, it may happen that some parts of the compiler have not been validated up to now (this is depicted as the hatched area in Figure 6). To validate it for the use cases of the project, appropriate test cases that cover the relevant missing parts of the code must be added and run and the results analyzed. Having done this, the compiler can be considered to be validated for use in the analyzed use cases because all relevant parts of the tool have been demonstrably checked.

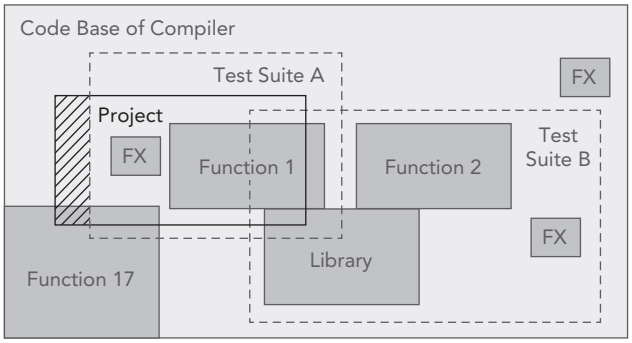


*Figure 6: Coverage-based metric for determining fulfillment of customer requirements*

This depends heavily on the architecture and realization of the compiler and on which coverage concepts have to be applied to get a reliable and reasonable metric. This could, for example, be code coverage with modified conditional/decision coverage for the C or C++ parts of the source code or "table coverage" for parsers that have been generated automatically from specified grammars. For some use cases, it might make sense to apply a metric to the input C code measuring the covered C grammar rules.

The benefits of this approach are that only the relevant parts of the compiler's code base have to be validated and that the validation results can be used for subsequent projects, leading to a further reduction of qualification efforts and making additional validation tasks superfluous.

### Realizing Coverage-Based Validation

As described in the previous section, the compiler can be validated for a use case with the aid of a metric based on the source code coverage of the compiler.

Figure 7 depicts a process for how coverage-based validation can be accomplished.

The following are the key elements of this process:

- The compiler source code covered by Wind River test suites is considered to be validated due to a thorough analysis.
- Wind River delivers a qualification kit to the customer that includes the following:
  - Starting qualification test suite
  - Instrumented compiler
  - Coverage comparator (which can also assess which parts of the compiler source code are not relevant for a use case)
  - Process description
  - User manual
- The customer runs the instrumented compiler on the qualification test suite, analyzes the results, and establishes validated coverage data.
- The customer applies the instrumented compiler in the intended use cases and obtains coverage data for the use cases.
- The customer compares the coverage data (use cases vs. qualification test suite) and assesses the relevance of the uncovered parts.
- The customer validates—alone or together with Wind River—the uncovered relevant parts of the compiler and extends the qualification test suite.

The outstanding advantage of such an approach is that it makes validating the compiler for special use cases feasible because efforts are already in an acceptable range in the beginning and may be applied to later projects. This holds true for the customer (by increasing confidence in the tool and increasing working knowledge) and for the tool supplier (by gaining broad information on usage, extending their own test suites and external quality assurance).

## Conclusion

A qualification test suite can be an approach for qualifying a compiler according to ISO 26262, if an appropriate metric is chosen for ensuring that the user's tool requirements are satisfied. All criteria required by the safety standard for tool validation fit into this approach. Furthermore, the feasibility of the proposed process with coverage-based validation has been used in a number of projects by Wind River and Validas AG with joint customers in the automotive industry. For such a process, an appropriate choice of the coverage concepts and an appropriate design of the test cases are crucial. Essentially, the test cases designed for closing the gap of non-validated parts of the compiler for a specific use case have to be devised so that the customer can be convinced of the correctness of the executable generated by the compiler.

The approach presented for a compiler validation goes beyond the state-of-the-art. Simply applying a test suite (whether commercial or proprietary) can, in general, give no evidence of correctness of the tool for a concrete use case. However, this can be achieved by using a coverage-based metric that leads to an optimal tailoring of the validation process to the use case and, thus, makes the validation efficient with respect to expenditure of time, personnel, and finances.

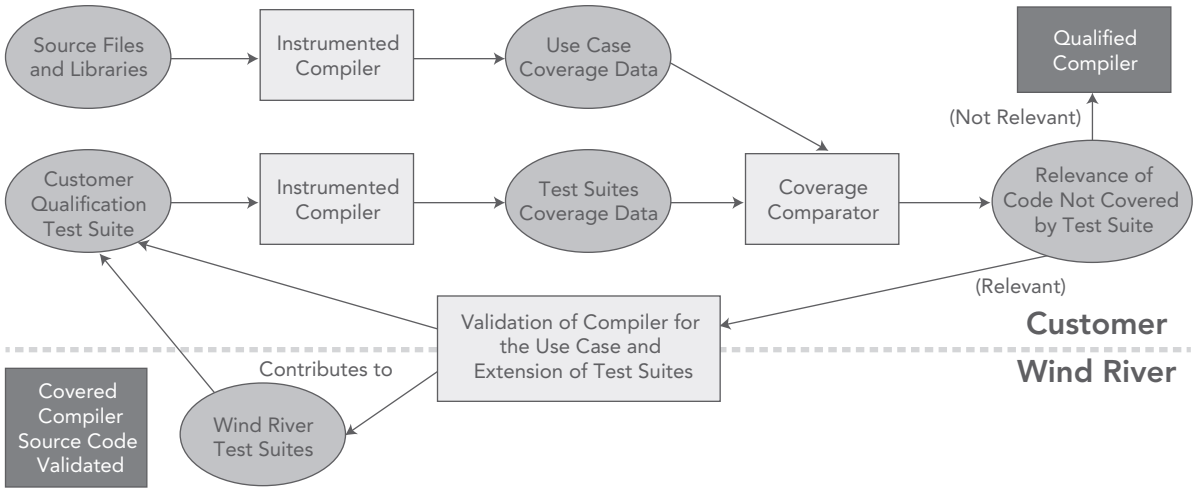


*Figure 7: Coverage-based validation process*