ISO 26262 - Configuration and Calibration test strategy

Author: Michael Stahl, Intel.   3 Jan 2019, v1.0
michael.stahl@intel.com / michael.m.stahl@gmail.com / www.testprincipia.com / LinkedIn

Acknowledgements

Thanks Simone Fabris, Maurizio Iacaruso, Gabriele Paoloni and Itamar Sharoni, from Intel and Mobileye, for their invaluable advice and review while writing this paper.

Terms and Acronyms

Term	**Description**
Configurable software	Code that contains constructs that allow generation of different binaries, based on parameters provided to the build and compile processes. Examples: #pragma statements; #IFDEF statements
Configured software	The binary created by the compilation of a configurable software. There would be a different “configured software” binary for every combination of configuration parameter values.
Lead platform	A HW platform that will be used for the bulk of testing. Lead platform is selected for each feature. The selection process is outlined in Appendix F - Lead Configuration selection
Configuration parameters	Parameters that are input to the build and compile process of the code. Different values of configuration parameters will create a different binary. See a discussion in Appendix E – Configuration Parameters and build scripts
Calibration parameters	Parameters that are input to the compiled software, and define the behavior of features associated with these parameters. The same binary (“configured software”) can be executed with different values of calibration parameters.
Independent calibration parameter	Configuration parameters that influence the resulting binary in the same way, regardless to the value of other configuration parameters
Dependent calibration parameter	Configuration parameters that influence the resulting binary in different way, depending on the value of other configuration parameters.
Hardware dependent parameter	See Platform dependent parameter
Platform dependent parameter	A configuration or calibration parameter that is related to certain HW platform(s). For example: a parameter that defines the use of certain sensor (e.g its sensitivity).

SW dependent parameter	A calibration parameter that is related to a feature that is HW-independent ("SW-only feature"). Note that in some cases a SW-only feature is made available only on certain types of HW . This would make the parameters associated with the feature HW-dependent, even though they are not really so.

Preface

This document outlines a strategy for testing what ISO 26262 refers to as "Configuration and Calibration data" (see Terms table for definition of these terms). The first pages (2 to 13) provide the strategy details, with minimal rationale. The rationale and additional support information is in the appendices.

Scope

For ISO 26262 compliancy, only configuration and calibration parameters that influence or interact with safety-relevant capabilities or features must have a clearly articulated test strategy. Note however that the strategy outlined here can also be applied to non-safety-relevant parameters.

The discussion covers the integration or feature level testing (black box tests) and not unit tests.

Overview

Consider a supplier that develops a HW/SW component to be integrated into a car. The software that goes with the component is already compiled, therefore it is a "configured software". The car OEM decides what calibration parameters will be used and what will be their settings.

At the vendor's validation level, we would like to ensure that any released configured software is providing the promised safety functionality. However, since it is unknown what calibration parameters will be used by the OEMs in the car, the only way to ensure the safety functionality in the configured software is to test all possible combinations of calibration parameter values on each binary version that the supplier ships. When the number of configuration and calibration combinations is low it is feasible to test all the possible combinations and remove the risk that the software will fail when set with an untested combination. But when there are many possible combinations, covering them all is more testing that can feasibly be done. We therefore need a justifiable test strategy that will keep the amount of testing that the supplier must do to a reasonable level while providing confidence that the shipped software will work as designed under any valid calibration parameter settings. The strategy below is such a proposal.

- ☑ The range of valid parameter values may be limited by an assumption-of-use (AoU). That is, while in theory a configuration or calibration parameter can take X values, the AoU narrows the number of values. Knowledge of the customers' intended use in the final product can be used as a mean to narrow the number of parameter values even further. Any reduction in the number of parameters' values increase the percent of combinations that can be feasibly covered.

Summary

In very high level terms, the test strategy is as follows:

Configuration parameters tests

1) Verify that the configuration parameters have valid values (review; no dynamic testing)
2) Select what HW-SW configurations will be used for testing the configured software and how much testing will be one on each HW-SW configuration. This selection is based on the configuration parameters values and the HW platforms details.

Calibration parameters tests

Calibration parameters tests are conducted on the HW-SW combinations selected based on configuration parameters values and HW platforms details.

1) Verify that invalid or out of range calibration parameters are rejected gracefully
2) Verify that the software under test behaves as expected, taking into consideration the calibration parameters values. This includes covering combinations of parameters.
3) Verify that the software can detect unintended changes to the calibration parameters' values, during run-time

The rest of this document dives into details of the strategy and to the rationale behind it.

Configuration parameters test strategy

Before diving into this section, check if you need to worry about it at all. If you provide a single binary of your code to the project it means you use a single set of configuration parameters values and you don't need to worry about testing various configurations. You do need to ensure that the parameters are valid (read Range and Validity Verification) but you don't need to worry about Configuration parameters Intent and Usage. If you decide to read that section anyway (some people just have free time on their hand), note that when you provide a single binary, all your parameters are considered "Static parameters".

High level approach

The ISO standard allows two options for validating the configurable software and configuration parameters [C.4.5]:

1. Verification of the configurable software + verification of the configuration data
2. Verification of the configuration data + verification of the configured software

The approach taken in this guide is option (2). To learn more about these two options and the rationale for picking option (2), see Appendix D – Configurable software verification.

Basic Properties of the Configuration parameters

For each configuration parameter, you should check that:

a) Valid values are accepted and invalid values rejected (the binary is not built).
b) The influence of the configuration values on the software binary and functionality is as required

The information about valid values and their influence should be defined in the architecture documents, the design documents or the Configuration Data Specification document.

Item (a) is covered in Range and Validity Verification below. Item (b) is covered later, in combination with the calibration parameters. Tests are executed on the HW-SW combination selected according to the directives in the Intent and Usage.

Range and Validity Verification

The first step is to verify that the value of each configuration parameter is in the range of valid values defined for this parameter in the architecture documents. This is done by a review of the Configuration data. The review should also ensure that if there is a dependency between parameters, only allowed combinations are used. You need to provide evidence that this review took place. Additionally, you need to provide evidence of proper configuration management and change management practices that prevent unintended changes to the parameters values.

There is no dynamic testing involved in this step.

[This section covers requirements C.4.1a,c,d; C.4.2]

Unit translation

In some cases, a configuration parameter is converted from real world engineering units (such as kilogram; Celsius etc.) to some abstract value that is the one used as the compiler directive. E.g. the memory size value of 2MB is translated to a configuration parameter of 0x0001. You must have tests in place to validate that any such translation is done correctly. If the translation is done by a pre-processing tool (before calling the compiler) – that tool needs to be classified and potentially qualified according to ISO rules (ISO 26262:2018, Part 8, Section 11). See also Appendix E – Configuration Parameters and build scripts .

[This section covers requirement C.4.1c].

Intent and Usage

Apart from verification that the parameters have valid values, tests must validate that the resulting binary is correct. For example: if parameter A enables feature X, we need to validate that feature X exists in the binary when parameter A is set, and is not in the binary when parameter A is not set. Additionally, the ISO standard also calls for verification that when a feature is enabled in the binary, it functions correctly. Together, the ISO standard calls this “the intent and usage”. [C.4.1b].

Since the behavior of the software is further impacted by calibration parameters, we explain below how to select the HW-SW combination on which testing will be done; the actual test strategy is postponed to Calibration Test Strategy.

Analysis

The procedure described below can be used to decide what HW-SW combinations you will test. It is based on the following conclusions (see

Appendix A – Analysis of configurations for detailed analysis):

- There are Static, Platform-dependent and Common configuration parameters
- There are four different relations between configurable-software and hardware platforms
- The approach for selecting HW-SW configurations to test on and the ensuing test strategy can be the same for all the four relations

Read Appendix A for justification of the above conclusions.

Static parameters test strategy

Ignore static parameters. Your test strategy does not need to make any special arrangements for coverage of static configuration parameters. You do need, of course, to validate the functionality that these parameters enable or disable, but it would be the same situation if the configuration parameters were hard-wired in the code.

Platform-dependent parameters test strategy

Assumption: The code for a feature in all the configured SW versions is more-or-less the same: much of it is common; the algorithms and approach for supporting the feature are the same; there are possibly some differences to accommodate different HW platforms[1].

For each feature:
- Select a Lead Configuration (see Appendix F - Lead Configuration selection)
- Continue to Calibration test strategy

Common parameters test strategy

The approach for common code is the same as for platform-dependent code. The only difference is that HW group A (see Appendix F - Lead Configuration selection) contains all the HW platforms. Once a lead platform is selected, any of the configured software binaries in SW group A can be used.

Combinatorial coverage of Configuration parameters

Parameters can be independent or dependent.

Independent parameter coverage

An **Independent parameter** influences the resulting binary in the same way, regardless to the value of other configuration parameters. Example: A parameter that adds the speed-limit feature to a cruise-control system. If set to 1, the code for the speed limit function is added to the binary; if set to 0 the code is not added.

Perform lead platform selection and continue to Calibration Test Strategy.

[1] If the assumption is wrong, treat each unique implementation of the feature as a separate feature

Dependent parameter coverage

Dependent parameters are those whose influence on the resulting binary is based on the combined values of a few parameters. Example: A configuration parameter defines the type of throttle control motor available in the target car. It can be type A or B. Another configuration parameter decides if the speed-limit feature would be added to the binary. The speed limit feature is added only when the throttle control HW is of type A.

This document does not propose a combinatorial coverage strategy for configuration parameters. Each combination of parameters generates a unique configured software binary and the strategy for testing this binary is as outlined for independent parameters: select a lead platform for each combination and continue to Calibration Test Strategy.

Commented [SM1]: ...alone (?)

[C.4.1d; C.4.2c].

- ☑ Rationale: In real-world projects we don't anticipate generation of many binaries. If a certain feature can accept many input combinations and behave differently based on that, it's fair to assume that this will be done by calibration parameters. The discussion of combinatorial coverage is therefore postponed to the calibration parameters test strategy. In the rare case where there is a plethora of binaries, generated by combinations of configuration parameters, the approach outlined for calibration combinatorial coverage applies. The end result will be that for certain features, you will have more than a single Lead Configurations.

Calibration test strategy

Calibration parameters that do not influence the safety relevant (SR) code, do not need to be covered by specific tests for ISO 26262 compliance. Note however that the strategy outlined here is valid to any calibration parameter and is recommended as a good testing strategy for non-SR parameters as well.

As mentioned in the Overview, the set of calibration parameter values that need to be covered may be significantly narrowed down by taking the published Assumption of Use into consideration.

Basic properties of the Calibration parameters

For each calibration parameter, you should verify by review or by testing that:

a) Valid values are accepted and invalid values rejected in a proper manner (hint: crashing is not a proper manner).
b) The influence of the calibration values on the software functionality is as required
c) The code has mechanisms that ensure the validity of the values before using them [Table 17 in C.4.10]. Note that this is not only for the initial use, but also in case the values are kept in memory and re-used during the program's execution.

The information about valid values and their influence should be defined in the architecture documents, the design documents or the Calibration Data Specification document.

Item (a) is covered in Range and validity below. Items (b) and (c) are covered later, in Intent and Usage.

Range and validity

The recommended test strategy is to design test cases using EC and BV test techniques (see Appendix B – Equivalence Class and Boundary value testing) and verify that parameters are accepted or rejected as expected.

[C.4.6a; C.4.6c]

Unit translation

Follow the approach outlined for Configuration parameters. [C.4.6c]

Intent and Usage

There are two types of calibration parameters:

- HW calibration parameters: parameters that define the behavior of features that are related to the target HW
- SW calibration parameters: parameters that define the behavior of features that are software-only features and are not dependent on the target HW

An example of a HW calibration parameter is an input that specifies the type of sensor available in the system. The information controls which software driver is used.

An example for SW calibration parameter is a parameter that defines the sensitivity of an algorithm for pedestrian identification. The input value defines how close to the car a pedestrian can stand before the pedestrian warning chime is sounded.

Additionally, parameters are split to dependent and independent ones (as defined in Combinatorial coverage of Configuration parameters).

HW calibration parameters (independent parameters)

HW calibration parameters function much like the “1-to-Many” case described in Appendix A. The software binary behaves as if the code related to other hardware platforms does not exist. The test strategy for the HW calibration parameters is as follows:

- Validate the feature’s functionality in full on the lead configuration[2].

- Run a reduced set of tests for this feature (also known as “touch testing” or “basic functionality testing”) on all the other binaries in SW group A. The goal of these tests is to show that the functionality is available in each binary and that it is “alive”. Use either the Lead Configuration or any other HW platform that can run the selected binary and supports the feature.

- Run a basic negative test on each of the configured software binaries is SW group B. The test can run on any HW platform that fits the configured software (from HW group A or B). The test validates that the feature indeed does not exist and that trying to use it results in appropriate response from the software.

Tests need to be executed with different values of the calibration parameter. The test design techniques to use are equivalence class partitioning and boundary value testing. See Appendix B – Equivalence Class and Boundary value testing for details.

SW calibration parameters (independent parameters)

The approach is the same as for HW calibration parameters. The only difference is that in most cases a single lead configuration can be used in testing all the SW calibration parameters.

Combinatorial coverage of dependent calibration parameters (HW and SW)

First, define the combinations you will cover. See Appendix C – Combinatorial coverage for details about options, techniques and considerations.

[2] If you have more than a single lead platform, split the tests among the lead platforms as makes sense according to the considerations that led to select more than a single lead platform. See details in Appendix F - Lead Configuration selection.

Out of the list of combinations you intend to cover, select the most important one(s). “Importance” can be decided following considerations similar to those of the selection of lead platform and lead binary: expected popular combinations in the field; combinations that are worst-case; etc.

The following test strategy is recommended for each feature that is impacted by the calibration parameters:

- On the lead configuration:
    - Full testing on the important combinations
    - Touch testing on the less important combinations
- On non-lead configurations:
    - Touch testing on the important combinations
    - Range and validity check on the less important combinations.

If there are invalid combinations (where the value of each parameter is valid, but the combination is invalid): Verify the invalid combinations are rejected. If there are many such combinations, apply combinatorial coverage considerations to decide what combinations to cover. Since these tests are usually fast, it may be feasible to cover all combinations.

Partially-supported features

If a feature is fully enabled on some platforms or binaries and only partially enabled on others, you can count on the tests done on the fully-enabled version and run only touch-testing on the partially-supporting binaries. You must run negative tests on each of the binaries with partial implementation to validate that the feature is alive, but the binary does not crash when attempting to use the un-supported parts.

Run-time validation of calibration data

ISO 26262:6-2018, Annex C.4.10 require that the code contains mechanisms for detecting unintended changes of safety-related calibration data. The following methods are proposed (Table 17):

Mechanism	**ASIL level**			
	A	**B**	**C**	**D**
Plausibility checks on calibration data	++	++	++	++
Redundant storage and comparison of calibration data	+	+	+	++
Calibration data checks using error detecting codes [a]	+	+	+	++

[a] Error detecting codes may also be implemented in the hardware in accordance with ISO 26262-5:2018.

Explanation of the mechanisms:

Plausibility checks: A mechanism to check that calibration data makes sense: e.g. the values are reasonable for the current state of the system and the values of other parameters in the system. These checks are done before the calibration data is used.

Redundant storage: The calibration data is stored in more than a single place and the value of calibration data is ensured to match in all storage locations, prior to using it.

Error detection mechanisms: Error correction codes or parity bits, implemented in HW or SW.

Validation of run-time validation mechanisms

A number of test techniques can be used to validate run-time validation of the calibration data.

Plausibility checks:

a) Using a debugger, enter breakpoint in the code before the calibration data is used. Change the value of a calibration parameter to a value that should be rejected by plausibility checks (invalid or invalid in the current context of the execution). Validate the code rejects this data in a graceful manner.

b) Add test code to the SW under test that will change the value of a calibration parameter at a certain point in time, while the software runs. An example may be to have the code read a value from a file, if the file exists. At a certain point in the test, create the file, with a calibration parameter value that should fail the plausibility checks. Add a log or some other mechanism to ensure the invalid calibration data was indeed read. Check the system's behavior when this happens.

c) If you know where (in memory) the system under test holds the calibration data and if you have access to it, change the data to an non-plausible value by an external process and check for the result. The benefit of this strategy is that you don't change the software-under-test and that you can run it under normal operating conditions.

Redundancy checks:

The above test techniques apply also for validity checks that are based on redundancy. Obviously only one instance of the configuration parameter should be changed. The one difference relative to plausibility checks is that in this type of check mechanism, the injected values can be plausible; but as long as they are not the same in all storage locations, the system should catch this and act accordingly.

Error detecting codes:

HW mechanisms should be tested by fault injection campaigns. SW mechanisms can be tested using the same injection techniques outlined above; the injected values must include an erroneous parity bit or some other error that will trigger the error correction code.

Other techniques can be thought of; the important feature of any such test technique is that it can change the calibration data at an arbitrary time to an invalid or un-plausible value.

Appendix A – Analysis of configurations

The first step in devising a test strategy for the configuration data is to analyze the configurations that are possible for a given code and decide what SW-HW combinations to test.

A reasonable assumption to make is that the configuration parameters can be split into two groups:

Configuration parameters types

Static: Configuration parameters that in theory can take a number of values but in fact are kept unchanged for all versions of the configured software. Examples:

- A #pragma directive that is always used
- A parameter that is used in #IFDEF clauses in the code and is always set to the same value
- An input switch to the compiler that is always used in the same manner
- Any parameter that is used with only a single value

Static parameters provide future flexibility to the developers or allow using a one-trunk development model – but don't provide differences in the resulting binary since they are always using the same values.

Platform-dependent: Parameters which take different values and therefore create different configured SW binaries. Values can be different so that the binary fits different HW platforms, or to just create different flavors of the SW (e.g. with or without certain features). For each HW platform, there may be one or more valid values for the configuration parameters. The result is that for each HW platform there is one or more viable versions of the Configured Software (CSW). Possible variations (and some examples) are below:

- **1-To-1:** Single CSW for each HW platform. Example: Assume the difference between HW platforms is the memory size. A configuration parameter indicate the memory size available on the target HW. The resulting binary is different, based on the memory size. Each HW has a specific CSW.

- **1-To-many:** Single CSW for a number of HW platforms. Example (continued from the previous example): If there are a number of HW platforms, differing in some manner (e.g. physical size) but with the same memory size, the same CSW fits a number of HW platforms.

- **Many-to-1:** A number of CSW for the same HW platform. Example: The same car HW can be used to provide cruise-control and speed limiter. The speed limiter is sold as an add-on to the base model. The add-on is activated by loading a different SW binary to the same HW. A configuration parameter controls whether the speed limiter code is added to the CSW or not.

- **Many-to-Many:** A number of CSW for a number of HW platform. Example: Consider a car that has the needed sensors for Adaptive Cruise Control (ACC). The SW binary for this feature is the same one that is used in a car that does not have the sensors. The SW, upon initialization, checks if the sensors exists. If not, it turns ACC off. Additionally, this SW comes in two binary flavors: with and without speed limiter.

Analysis

The four versions of platform-dependency can be schematically drawn as below:

(CSW = Configured Software; of course there may be more than just two SW or HW versions).

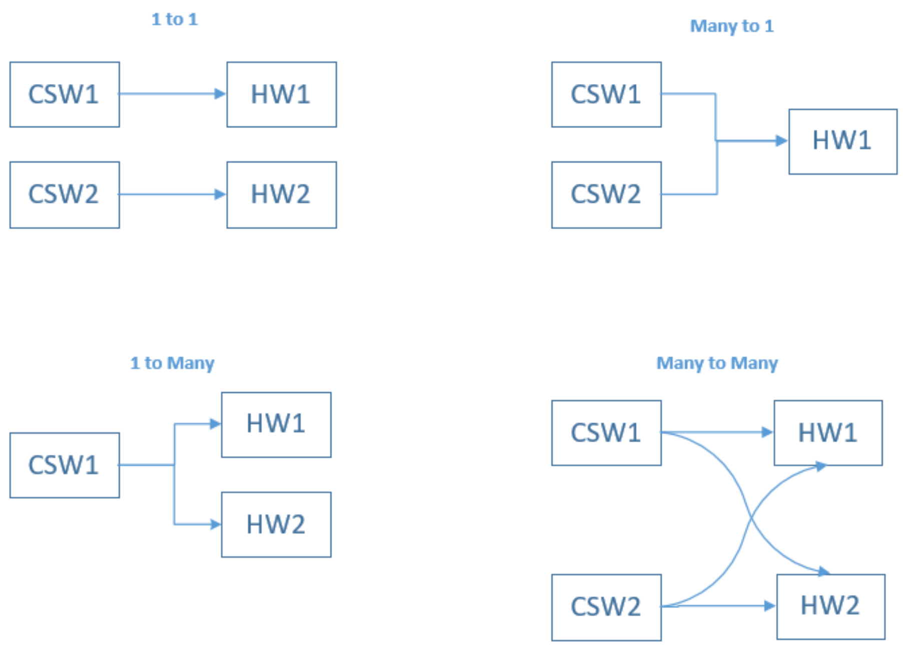

Figure 1 Four platform-dependent cases (initial view)

In reality, the difference between CSW1 and CSW2 is usually not that large. Each configured software can be seen as made of two parts:

Common code:

- The code is exactly the same
- The code's behavior is exactly the same for all the target HW variations

HW-specific code:

- The code's behavior is different for different target HW

- The code may or may not be exactly the same (when the code is the same, different behavior for different HW configurations is a result of the code's internal logic)

This means that the four versions of the Platform-dependent case can be schematically drawn as below:

(HWS = HW-Specific).

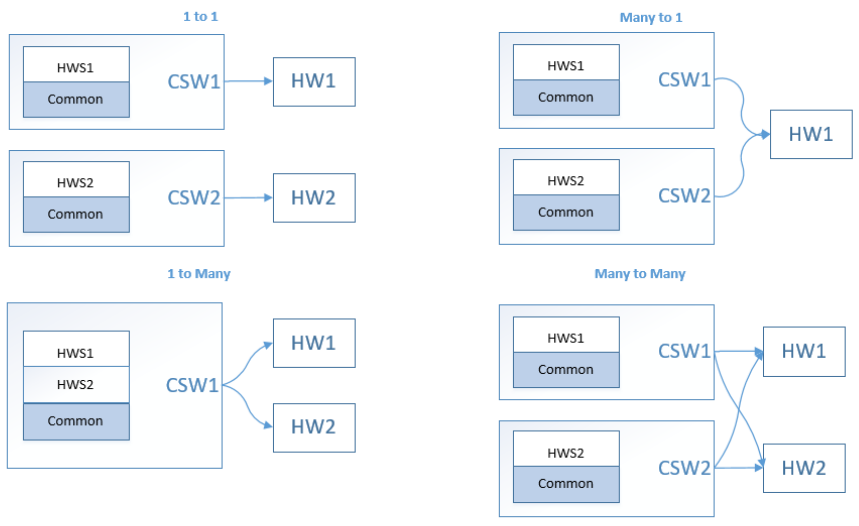

Figure 2 Four platform-dependent cases (common VS HW-specific blocks)

Figure 2 has a number of blocks with two entry or exit arrows. These can be expanded to have only a single arrows between blocks:

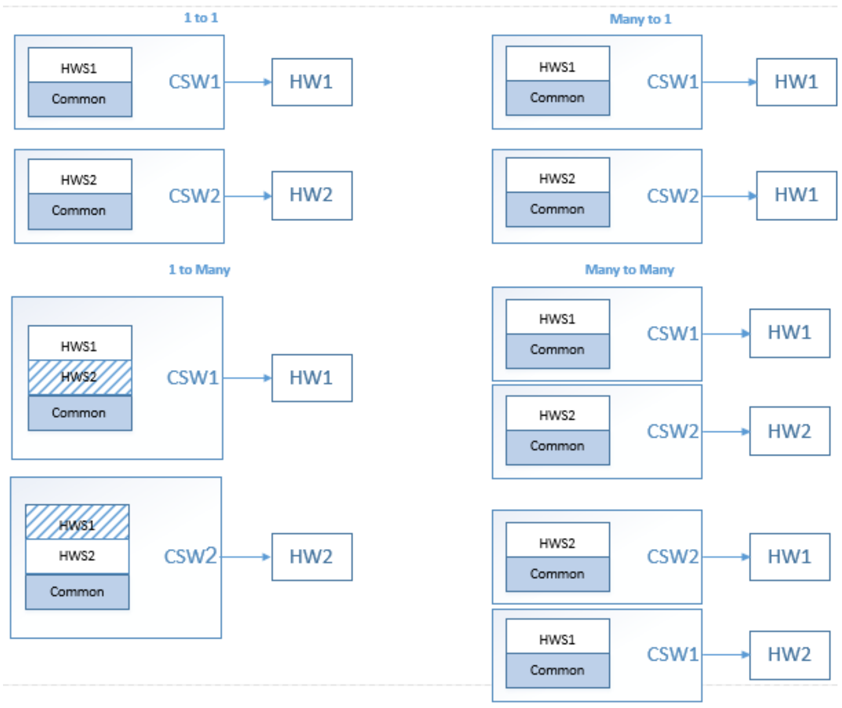

Figure 3 Four platform-dependent cases (expanded view)

Figure 3 shows that all the four variations of the Platform-dependent case are very similar. The many-to-1 case is exactly as the 1-to-1 case, with the difference that HW1 = HW2. The 1-to-many case is also the same as the 1-to-1 case, under the assumption that only the relevant HW-specific part of the code is active for each HW. The first pair of blocks on the many-to-many case is identical to the 1-to-1 case; the 2nd pair is identical once the definition of CSW1 and CSW2 is switched.

All this leads to the conclusion that for each feature, all the types of SW to HW dependencies can be described as a multiple set of matched SW-HW pairs. The HW specific part is the part of the code that is HW dependent. The Common part is the code that is the same for all the configured SW cases. Some features are solely in the HW-specific part. Some are solely in the Common part; some are split: part of the code is common across all configured SW and some is different for each HW.

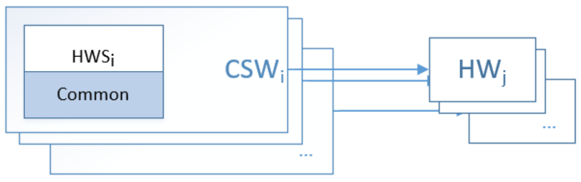

Figure 4 HW-SW combinations

The tests for each of the HW-SW pairs would therefore be pretty much the same, with attention paid to what CSW and what HW we refer to.

It also means that the configuration parameters can be split into two categories:

- **Platform-dependent parameters**: parameters that influence platform-dependent features
- **Common parameters**: parameters that influence platform-independent ("common") features.

Note that platform-dependent feature does not necessarily mean a HW dependency. There may be features that are software-only features but for business considerations are available only on certain HW platforms. For the sake of the above analysis, they too are included in the platform dependent features and are possibly controlled by platform-dependent parameters.

Appendix B – Equivalence Class and Boundary value testing

The text below assumes that the test techniques of Equivalent Class Partitioning (EC) and Boundary Values (BV) are known to the reader.

When EC and BV testing is mentioned in the text, it means as following:

- Perform an equivalence class partitioning and define the valid and invalid equivalence classes for the parameter. Values that are disabling a function are considered valid values.
- For each EC, select representatives of the EC:
    - If the valid EC is a closed range, select the high and low boundary values.
    - If the valid EC is one-sided range (e.g. >1), select the boundary value and one more value away from the boundary
- For each valid EC representative value, run a test that validates:
    - The valid value is accepted
    - The value influence on the code's functionality is as required.
    - If the influence on the code is such that a certain functionality is NOT in the code, run tests to validate:
        - The functionality is indeed not in the code
        - The system behaves reasonably when use of the non-existing function is attempted
- For each invalid EC representative value run a test that validates:
    - The value is rejected by the software in a reasonable manner (e.g. with an expected, legible message or error code; abort; revert to a default)

Notes:
- If the parameter is an enumeration, each enumeration value is an equivalence class (EC) by itself
- If there are "interesting" values in a range (e.g. the value 0) – test these as well

- When setting a certain parameter to a boundary-value, set the values for all other to a valid, non-boundary values and preferably vary these settings on each test. Set enumerated calibration parameters to any valid value.
    - The above does not apply to combinatorial testing, where the values of each dependent parameter is defined by the covered combination – so it is possible that more than one parameter will be at the boundary.
- When testing invalid EC for a certain parameter, all other configuration parameters must be set to nominal (non-boundary), valid values.
- When there is a dependency between parameters, the definition of valid and invalid values may change according to the combination of parameter values. E.g. Parameter A can take the values of 1,2,3 if parameter B is set to 1, but only values 1,2 if parameter B is set to 0. This needs to be taken into account when specifying tests of range and validity. Since these tests are normally short, it is recommended to test all possible combinations. See a discussion of combinatorial coverage in Appendix C.
-

Appendix C – Combinatorial coverage

Perform an equivalence class partitioning and define the valid and invalid equivalence classes (EC) for all the parameters that influence each other. For each EC, select representatives of the EC (see more details in Appendix B).

Depending on your case, decide the level of combinatorial coverage to provide. This can be anything from Each Choice (1-way coverage); Base Choice; All-pairs or higher combinatorial coverage levels. It is worth noting that there is some evidence in research that errors do occur (albeit very seldom) at up to 6-way combinations[3,4] . Microsoft recommend up to 6-way combinatorial coverage[5]. It therefore seems prudent, at least for safety-related features, to cover all combinations up to 6-way coverage. This may not be feasible though if your parameters take a large set of values.

Other options include various combination of coverage level. For example, you can choose a different combinatorial coverage between lead platform on non-lead (e.g. 6-way for lead; base choice for non-lead).

A simple and much-recommended tool for generating combinatorial test cases is PICT. This tool allows you also to specify the combinatorial coverage, define certain constraints so that only valid combinations are generated and control other aspects of the generated test cases. It also has the option of manually specifying certain combinations of parameters you want to make sure are tested. It is recommended that the expected common combinations of parameters are covered by your tests.

Another approach to take is to test the code with random settings of the dependent parameters (or even all parameters). There is some evidence in the literature that random testing at the same level of planned combinatorial testing (that is, with about the same number of test cases) yields equivalent results in term of finding bugs[6]. Random testing has the benefit of ease in generating test cases and the fact that each test cycle is a bit different than the other. On the negative side, it is less controllable (if there are specific combinations you want to hit, random test cases won't guarantee it); you must log, for each test cycle, the actual values used in testing, for repeatability sake and for test-log requirements of ISO.

[3] Software Fault Interactions and Implications for Software Testing; Kuhn, Wallace, Gallo; http://ieeexplore.ieee.org/abstract/document/1321063/?reload=true
[4] http://mse.isri.cmu.edu/software-engineering/documents/faculty-publications/miranda/kuhnintroductioncombinatorialtesting.pdf
[5] How we test software at Microsoft; Page, Johnston, Rollison; 2009 edition; p.111
[6] Comparing the Fault Detection Effectiveness of N-way and Random Test Suites; Schroeder, Patrick J.; Bolaki, Pankaj; Gopu, Vijayram; http://ieeexplore.ieee.org/document/1334893/

Appendix D – Configurable software verification

This text is a summary of a discussion on Planet Blue:
https://soco.intel.com/message/5274340?et=watches.email.thread#5274340

In ISO 26262:2018, Appendix C, section C.4.5 the standard allows two options for validating the configurable software and configuration parameters:

The standard's wording:

> C.4.5 For configurable software a simplified software safety lifecycle in accordance with Figures C.2 or C.3 may be applied.
>
> NOTE A combination of the following verification activities can achieve the complete verification of the configured software:
>
> a) "verification of the configurable software",
>
> b) "verification of the configuration data", and
>
> c) "verification of the configured software".
>
> This is achieved by either
>
> - verifying a range of admissible configuration data in a) and showing compliance to this range in b), or
> - by showing compliance to the range of admissible configuration data in b) and performing c)

First, understanding of the wording used in C.4.5 a, b, c:

- All are using the word "verification of…"
- In (a) and (c), this means dynamic testing (on top of the architecture, design, code reviews that should also take place)
- In (b) this is a review of some sort, to ensure the right values are given to parameters. No dynamic testing involved in (b).

This is based on the assumption that some real testing must take place on the configured SW (can't base everything on reviews).

The standard allows two options for validating the configurable software and configuration parameters:

1. Verification of the configurable software + verification of the configuration data (C.4.5 a + b)
2. Verification of the configuration data + verification of the configured software (C.4.5 b + c)

But what exactly does this mean and when does it make sense to use each option?

Option (1) calls for first testing that the code is correctly generated and correctly functions for all possible combinations of the configuration parameter values (at least those that are associated with safety requirements). The second step is to verify that the actual configuration parameters used to create released binaries are within the valid range of the configuration parameters values.

Option (2) calls for first deciding the values of the configuration parameters and making sure they are within the valid ranges defined in the architecture; then generate the resulting binaries and test those binaries for proper functionality.

Following this explanation, it seems option 1 will always call for more effort than option 2. So why use it?

There are three cases where option 1 makes sense. The first one is when there are configuration parameters that accept a range of values (as opposed to most configuration parameters I have seen, that take specific enumerated values). In such case, validation of all values in the range would be impractical (too many tests). You'd treat this case as you would normally treat a test of an input parameter with a valid range. E.g. use equivalence class partitioning combined with boundary value test technique to select the test values. In the case of a configurable software, you will create two binaries, with the two boundary values of the configuration parameter. You will test both these binaries (this would be step C.4.5a) and then check that the actual configuration parameter used is within the valid range.

A second case (which is really just a variation of the first case) is when some data is embedded into the binary. For example, a binary may be compiled with the calibration data of a sensor. Different calibration data generate a different binary. This too is a case of a parameter that can take a range of valid values.

The third case is for open-source code, or for code that is given to the end user and the end user compiles the binary. In these cases you can't know in advance what configuration parameters will be used, so you need to test the configurable software.

If all the configuration parameters are enumerated (not a range), it will always be less work to go with option 2. This is the reason why the approach taken in this guide is option (2).

Appendix E – Configuration Parameters and build scripts

This text is the "bottom line" of a discussion on Planet Blue:
https://soco.intel.com/message/5273914?et=watches.email.thread#5273914

Compilation of non-trivial software is usually accomplished by more than a single compilation command. Use of tools such as CMAKE and of scripts is common. The build process may also contain sections of automated code generation. Such a build process involves many individual calls to utilities, compilers and linkers, each time providing command line arguments. These arguments may point to files that contain parameters that are also impacting the binary-generation flow.

In such an environment, what are the "configuration parameters"? Do we need to check all the parameters used in each command line?

The following is based on a discussion with Gabriele Paoloni:

- The parameters used when calling the build script are Configuration Parameters
- The build script is a Tool.

This means:

- Only the parameters provided to the build script need to be evaluated as in ISO 26262:2018 Appendix C. They need to be well-documented, including use, valid ranges, interactions etc.
- The build script needs to go through the process of tool evaluation and possibly qualification (if concluded to be TCL2 or TCL3).
- The build script and all associated files must be treated as code and put in a Configuration Management tool.
- Since any change to the build script or the associated files may invalidate the tool qualification, they must all be put under strict Change Management control. Each proposed change to the script or associated files must be evaluated to decide if the change calls for re-qualification of the build script.

Appendix F - Lead Configuration selection

The Lead configuration is the combination of a Lead Platform and a Lead Binary. It is selected on per-feature base. Most of the tests of the feature will be conducted on the Lead Configuration.

Lead Configuration selection process:

For each feature:

- Split the available configured software versions into two groups:
  - A group that supports the feature (SW group A)
  - A group that does not support the feature (SW group B)

- Split the available HW platforms according to the previous split:
  - Platforms that support the feature (meaning: at least one of the binaries in group A can run on this platform; the feature is enabled on this platform) (HW group A)
  - The rest of the platforms (HW group B)

- Select a Lead Configuration
  - From HW group A, select a Lead Platform, following Lead Platform selection
  - From SW group A select a Lead Binary, following Lead Binary selection

  Selection of the lead platform and lead binary may be an iterative process, as the considerations for platform and binary may contradict each other (e.g. the worst-case platform does not support what seems like a logical lead binary). The end result may be more than a single Lead Platform for the feature.

Lead Platform selection

The selection of a Lead Platform should take into account the following considerations:

- If a feature is fully enabled on some platforms and only partially enabled on others, give priority to the platform where the feature is fully enabled.

- Give priority to the platforms expected to be most common in the market. For example: The expected market for platform A is 1Meg units, and 10K units for platform B; platform A is a better candidate for lead platform.

- Give priority to the most sensitive platform. “Sensitive” is feature-related and should pose the worst case scenario for the feature.

    - Example 1: For a feature that is time-constrained, select the lowest-performance platform as the lead platform.
    - Example 2: For testing throughput of real-time sensor data processing, select a platform with many sensors feeding data at high rate; for testing the accuracy of the processing results, select a platform with limited number of low-end sensors.

  From the above it follows that the lead platform may be different for each feature – depending on the requirements of each feature.

- Consider changing the lead platform between test cycles. This reduces the risk of missing a platform-dependent issues even further. If you have more test cycles than HW platform flavors, this can ensure each platform runs at least once as the lead platform; more test cycles can be allocated to the worst-case platforms / high market-share platforms.

- If you have performance requirements for each HW platform, there is no way to work around that; you will need to run the performance tests on each platform.
    - Corollary: Safety requirements that are time-dependent must be tested on all HW since the timing is different (note that this is correct even when all the HW platforms uses the same binary).

The above considerations may contradict each other. As a result, you may find that for a specific feature you must select more than a single lead platform. For example, this may happen when a certain feature is enabled only partially on some platforms, but these platforms pose a worst-case for the feature.

Lead Binary selection

The selection of the Lead Binary follows the selection logic of the Lead Platform.

The selection of a Lead Binary should take into account the following considerations:

- The binary can run on the selected Lead Configuration.

- If a feature is fully enabled on some binaries and only partially enabled on others, give priority to the binary where the feature is fully enabled.

- Give priority to the binaries expected to be most common in the market.

- Give priority to binaries that accept the most diverse set of calibration values that affect the behavior of the feature-under-test. “Diverse” includes the number of calibration parameters and the number of different values (or range) that calibration parameters can take.

The above considerations may contradict each other. As a result, you may find that for a specific feature you must select more than a single lead binary. For example, this may happen when a different aspects of a certain feature are enabled on different binaries. In such case, you will need to select more than one lead binary. Each will be used to test (at least) the part of the feature that is enabled only on it and to negative-test the part that is not enabled. If the different binaries cannot run all on the same HW platform, you will also have to select more than a single lead platform.