Distributed Real-time Architecture for Mixed Criticality Systems

Variability Analysis and Testing Techniques for Mixed-Criticality Systems

D 4.3.1

Project Acronym	DREAMS	**Grant Agreement Number**		FP7-ICT-2013.3.4-610640	
Document Version	1.0	**Date**	2015-07-30	**Deliverable No.**	D 4.3.1
Contact Person	Franck Chauvel	**Organisation**		SINTEF	
Phone	(+47) 4790 7838	**E-Mail**		franck.chauvel@sintef.no	

Contributors

Name	Partner
F. Chauvel	SINTEF
A. Vasilevskiy	SINTEF
T. Trapman	ALSTOM
F. Eizaguirre	IKERLAN
F. Eizaguirre	IKERLAN
F. Ruiz	FORTISS
S. Barner	FORTISS
A. Diewald	FORTISS

Table of Contents

Executive Summary

This deliverable (D 4.3.1) describes how variability is resolved in the DREAMS model-driven process. It encompasses the specification, resolution and realization of variability, as well as it interaction with extra-functional properties such as safety or power consumption. The envisioned solution combines variability modeling and evolutionary optimization (both described in D 4.1.2) into a tool chain to be later implemented in D 4.3.2 and D 4.3.3.

The key contributions of this deliverable are:

- The specification of the tool chain supporting the resolution of variability in the DREAMS model-driven process;
- The assessment of the existing tooling and the identification of needed improvement for the BVR tool, the DSE tool and the safety constraint checker.

1 Introduction

To foster the development of mixed-criticality systems (MCS), the DREAMS project advocates a model-driven process. This process includes three successive refinements to transition from abstract specifications to configurations tailored for heterogeneous platforms.

In Work Package 4, we develop the tools supporting each of these steps. Task 4.3 first resolves the variability in the system specification. The resulting models are then refined by Task 4.1 into platform specific models, eventually converted in to platform configurations in Task 4.2. Ultimately, demonstrators evaluate these tools with the support of Task 4.4.

In the description of work (DoW) [1], Task 4.3 defines *"means to resolve the explicit variability's*". Mixed-criticality systems share a common basis, but systems always vary from one another. They vary not only in the functions they offer, but also the way these functions are built. While these variations are opportunities for designers, they remain constrained by others requirements specific to mixed-criticality systems: reliability, energy consumption, security, safety, etc. These requirements limit the space of possible solutions, and one must verify the relevance of candidate solutions using appropriate verification techniques such as simulation, testing or scheduling analysis.

This report – Deliverable 4.3.1 – is the first outcome of Task 4.3. It aims to "*provide an overview and assessment of different analysis and testing techniques that may be useful for mixed criticality systems with explicit variability model*". It also aims to "*specify potential improvements that should be done on these techniques*" (cf. DoW [1]). The related tools will be delivered in the two subsequent deliverables, namely D 4.3.2 and D 4.3.3.

Our approach distinguishes between *business* and *technical* decisions, which resolve variability in what the system does and how the system does it, respectively.

Business decisions govern the features of interest, such as "*all communication shall be encrypted*". These decisions are first captured in a *variability model*, which documents how these decisions are realized into the system specification, initially defined in D 1.4.1 [2]. We build product lines engineering tools, discussed in D 5.5.1 [3] to model variability using feature models, and to generate consistent sets of decisions using product sampling techniques.

Technical decisions govern the implementation of the features, for instance "*the AES RC6 algorithm shall be used for encryption*". These decisions are injected into the resulting specifications by evolutionary optimization techniques, developed in Task 4.1 and documented in D 4.1.2 [4]. Throughout this second refinement, extra-functional requirements are enforced using specific verification techniques such as the safety constraint checker.

The remainder is organized as follows. In Section 2, we first recall the basics of variability modeling, and we then elaborate on the challenges of verifying extra-functional requirements under variability, before devising enhancements needed for Task 4.3. In Section 3, we detail the procedure we propose to realize the variability in DREAMS specification. It starts with reviewing the DREAMS architecture models, and then describes how variations in these specifications may impact extra-functional requirements from mixed-criticality systems. Section 4 unfolds this process on an excerpt of the wind power demonstrator before Section 5 concludes.

2 Verification Under Variability

By contrast with technical decisions, business decisions are captured using techniques from product lines engineering. We recall below the ideas of product lines engineering and how they apply to the design of MCS.

2.1 Variability in a Nutshell

Software product line engineering (SPLE) fosters the development of families of related products by capitalizing on their commonalities, ideally in all phases of the development process. In this setting, *variability modeling* captures at a high-level what varies between these products (resp. what remains unchanged) using the concept of *feature*. A feature stands here for "*a unit of functionality of a software system that satisfies a requirement, represents a design decision, and provides a potential configuration option*" [5].

In DREAMS, variability modeling is supported by the BVR tool [6]. By contrast with other approaches directly deriving complete products, BVR works on modeling languages (e.g., UML [7, 8], SysML [9], AutoFOCUS3 [10]) and assumes the existence of transformations, which derive the final product from its associated model. In addition to the variability model, BVR distinguishes between the *resolution* of variability and the *realization* of features. The resolution of a variability model specifies which features are needed for inclusion in a particular product. The realization of a feature describes how to inject this feature into the model of the product. In a nutshell, the combination of variability, resolution and realization permits to derive the models of any products in a given product line. In addition, BVR supports checking that a given resolution matches the constraints of the product line, as well as checking for the existence of a valid product in any product lines. We refer the reader to D 1.4.1 [2] for a detailed description of the BVR tool and its capabilities. Additionally, Section 3 of D 5.5.1 [3] provides an extended overview of variability modeling covering both BVR and other approaches.

2.2 Variability Analysis

2.2.1 Mixed-Criticality Systems

As stated in the EC Workshop on MCS [11], "*Increasingly, there is a move towards integration of critical and non-critical functionality on the same platforms to create mixed criticality systems. The differences can arise from the needs of timeliness where mixed-criticality might describe a mixture of soft, firm and hard real-time applications integrated into one system*". According to this report, criticality can be understood in two main axes:

- *Time-critical systems focus on the availability of outputs within predefined intervals, typically indicated by a hard deadline.*
- *Safety-critical when considering the required degree of safety. Mixed-criticality might describe applications that are classified according to different safety levels (IEC 61508 [12-19], DO-178B [20], D0-254 [21], ISO 26262 [22-31]). Here the property of function integrity is especially important, i.e., functions must either provide the correct output according to the specification or indicate its failure. No deviation from specification is permitted, not even temporarily. This contrasts with time-criticality, which does not automatically imply the necessity of function integrity all the time.*

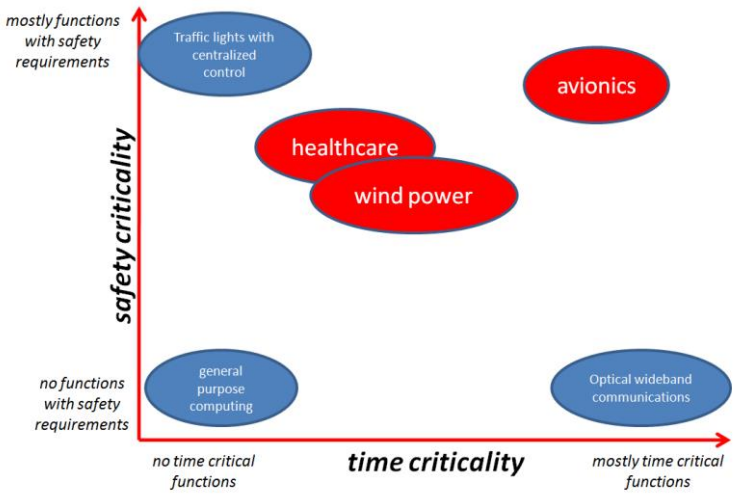

Figure 1. Mixed-criticality main axes – adapted from [11].

This deliverable summarizes the state-of-the-Art in variability analysis techniques. We emphasize here safety and timing analysis techniques but we shall also account for other extra-functional properties, especially reliability and energy consumption.

2.2.2 Overview of Analysis and Testing Techniques for Software Product Lines

A software product-line of MCS is a set of similar products that still differ from one another. To produce a given product, the user has to select the desired features (resolution) and the associated tool generates the product (realization).

As with any other software engineering tool, a software product-line must provide means to guarantee that any product is verified and validated. As the number of features increases, the number of possible products grows exponentially. Therefore, on the one hand, we need analysis tools to verify and validate products and, on the other hand a way to cope with the exponential number of possible products.

As proposed in [32], software product-lines encompass (i) product-line implementation techniques (ii) strategies for product-line analysis (iii) tool strategies and (iv) software analysis techniques. The following sections describe these dimensions.

2.2.2.1 Product-line Implementation Techniques

Implementing a product-line consists in managing the product variety. There are two main implementation techniques, namely annotation-based techniques and composition-based techniques.

In *annotation-based* techniques, source code fragments are annotated with features or combinations of features. Depending on the selected features, some code fragments may be included in the final product or may be eliminated. In *composition-based* techniques, the product results from the composition of separate executable units (e.g., modules, components, services), whose inclusion is governed by the selected features.

In DREAMS, such units are software components, hardware platform elements, system software elements (e.g., hypervisors, partitions), safety compliant items and model elements like bus connections or their properties. A set of meta-models represents the composition of the product-line and an external explicit variability model is in charge of eliminating elements not present in the final product.

2.2.2.2 Strategies for Product-Line Analysis

Product-line analysis is the process of scrutinizing product lines and the set of products they define. The related analysis techniques focus either on a subset of representative products covering as much

errors as possible, or take into account the variability and are able to analyse the family of products as a whole (these techniques are called family-based).

2.2.2.3 Strategy of the Tool

There are three main approaches when implementing a product-line analysis tool, namely product-based strategies, variability-aware strategies and the variability-encoding strategy.

In *product-based* analysis, the analysis techniques are applied to the generated products. In *variability-aware***,** the tooling is able to handle the product-line as a whole, without deriving any final products. For example if the variants are represented with annotations (e.g., "#ifdef" directive in C/C++), these techniques evaluate the consequences of different sets of features. Finally, using *variability-encoding***,** variability is encoded using conditional branching, so that the whole product-line is a single complex executable product.

2.2.2.4 Software Analysis Techniques

These techniques are classified into three categories (i) *testing*, (ii) *verification* and (iii) *further analysis*, plus an orthogonal category, *sampling*, for the techniques that are applied to only a subset of products (generated according to a given coverage of features criterion).

2.2.2.4.1 Testing Techniques

Testing techniques imply *executing* the product to assess some properties. Testing techniques include test-case generation, product sampling, and family-based testing.

The use of *test-cases* is common practice in Software Engineering. However, in the case of product-lines, it may be necessary to generate customized test cases for each product, because features affect what can be tested. Test-cases are in turn divided into *unit-tests* (test individual function/components), *integration tests* (to test composition of units), and performance tests (to test performance properties such as response-time, accuracy, etc.).

Techniques of *product-reduction* are able to reduce the number of products needed for a given test-case. Finally, *family-based* testing techniques can execute all products in parallel (for example computing multi-value data for combination of features).

2.2.2.4.2 Verification Techniques

By contrast with tests, verification techniques analyse the product *without executing* it. Verification techniques encompass type-checking, static analysis, model checking, theorem proving and consistency checks.

Type checking makes sure that the source code of every possible generated product compiles with no errors. *Static analysis* operates on compile-time and, without execution, can predict run-time behaviour/values and therefore can make some checking.

Software-model checking translates the product-line, when possible, into a state machine that is easier to analyse (the graph represents the whole family of products and different combinations can easily be checked). *Theorem proving* first translates the program and its specifications into logical formulas and uses deductive techniques to check its correctness.

Finally in *consistency checking*, the consistency of artefacts corresponding to different features is tested, as for example, if all involved artefacts are presents, if dead or superfluous source code is produced under some combinations of features, etc.

2.2.2.4.3 Further Analysis Techniques

Beyond the verification and testing of the functions provided by a product, extra-functional properties can be analysed, as well as metrics reflecting the quality of its source code.

One of the goals of DREAMS platform is to optimize some extra-functional properties (e.g., energy consumption, response time) while verifying some others (safety claims, reliability). The analysis

techniques are specific to each extra-functional meta-models of DREAMS and are a main research challenge.

2.2.3 Variability Analysis in DREAMS

As shown before, multiple analysis tools and techniques exist (either as research prototypes or as commercial tools) to analysis the correctness of the code generated from the product-line. An extensive state-of-the-Art can be found in [32] and some techniques are applicable to DREAMS.

However, specific extra-functional analysis techniques are tailored for MCS and have to be integrated in the DREAMS model-driven process. This deliverable focuses on three main extra-functional properties, namely safety, timing, and reliability. In addition, we will cover energy consumption, though it does not characterize all MCS.

As already mentioned above, testing implies execution while verification does not imply execution. Any testing or test-case is specific to the uses-cases and therefore, it is difficult to add testing as a generic tool to the product line. However verification techniques can be developed for extra-functional properties so that the product-line can filter out non valid products.

2.3 The Feature Interaction Problem

The construction of software product lines is greatly enhanced by designing loosely coupled parts, so that different products can emerge from assembling different set of parts. As for verification, the systematic verification of every part however fails to guarantee the proper behavior of assemblies. Parts interact in unforeseen ways and, in turn, assemblies behave unexpectedly and eventually fail. In 1996, the European Ariane 5 space launcher exploded after about 40 s. of flight because of such an erroneous interaction [33], which costs 500 000 000 US dollars. This *feature interaction problem* is characteristic of complex systems [34] and exacerbates the issue of their verification.

To be effective, verification thus has to cover every possible interaction. Empirical studies [35-37] show that the sole test of interactions between pairs of parts – so called 2-wise interactions test – already improves defect detection from about 50 % to 70 %. As shown on Figure 2, further including 3-wise interactions would then detect about 95 % of defects. All defects would eventually be found by investigating up to 6-wise interactions. Unfortunately, the number of possible interactions grows exponentially with the number of features. This is a key obstacle to the verification of industry-sized systems.

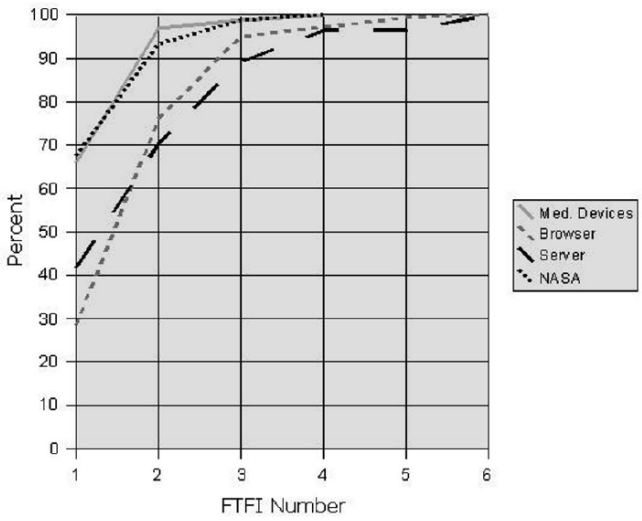

Figure 2 Percentage of fault due to interaction – borrowed from [36].

The verification of software product lines is impeded by this feature interaction problem. The number of systems that can be derived from a product line grows exponentially with respect to the number of features. Deriving and verifying every single product is thus not feasible in practice.

2.4 Combinatorial Interaction Testing

Combinatorial interaction testing (CIT) [38] addresses the feature interaction problem, at both the operation level and at the system level.

At the operation level, interactions occur between parameters value. As unit-testing is cheap, combinatorial testing thus produces all the test cases needed to cover these possible interactions. Yet, most parameters accept an infinite – or at least a huge – number of possible values (e.g., number, text string, byte fields). These values are therefore grouped into categories, such as positive, negative or null for numbers. The generated test suite ensures that every possible combination of categories is secured.

At the system level, interaction occurs between configuration options: hardware components, operating systems, Internet browsers, network protocols, encryption algorithms, character encoding, etc. The number of possible integration tests is too large to be practical, because running a single integration test is already expensive and time consuming. The objective is therefore to select a subset of these integration tests, which secures a maximum of interactions.

In practice, CIT requires the computation of a *covering array*. This array associates a set of test cases along with the options – or categories – they exercise. Figure 3 shows a covering array for the Eclipse IDE[1], where test cases are associated with the plugins they execute. Only 22 test cases are needed to cover all possible 2-wise interactions between plugins. Although computing a minimal covering array is resource-consuming, recent advances [39, 40] now make CIT practical for industry-sized systems, beyond pair-wise interactions.

As for product lines where interactions occur between features, CIT goes beyond the mere validation of every possible product, and verifies a product line as a whole. In his PhD dissertation [40], Martin F. Johansen proposed a faster algorithm for computing covering arrays on large product lines. This algorithm is available in the SPLCA tool and integrated in the BVR tool. In BVR, covering arrays also account for the constraints embedded in the product line. They are transformed into a logical formula, which permits to derive a minimal set of products that includes a large number of feature interactions. The interested reader may refer to D 5.5.1 [3] for a more comprehensive treatment of product line testing.

[1] See https://eclipse.org/

Feature\Product	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
EclipseSPL	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
WindowingSystem	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Win32	-	-	X	-	-	X	-	-	X	-	-	-	-	-	X	-	-	-	-	-	-	X
GTK	-	X	-	-	-	-	X	-	-	X	-	-	-	X	-	-	X	-	-	-	-	-
Motif	-	-	-	-	X	-	-	X	-	-	-	X	-	-	-	-	-	-	X	-	-	-
Carbon	-	-	-	X	-	-	-	-	-	-	X	-	-	-	-	X	-	X	-	-	-	-
Cocoa	X	-	-	-	-	-	-	-	-	-	-	-	X	-	-	-	-	-	-	X	X	-
OS	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
OS_Win32	-	-	X	-	-	X	-	-	X	-	-	-	-	-	X	-	-	-	-	-	-	X
Linux	-	X	-	-	X	-	X	X	-	X	-	X	-	X	-	-	X	-	X	-	-	-
MacOSX	X	-	-	X	-	-	-	-	-	-	X	-	X	-	-	X	-	X	-	X	X	-
Hardware	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
x86	X	-	-	X	X	-	-	X	-	X	X	X	-	-	X	X	-	X	X	-	-	-
x86_64	-	X	X	-	-	X	X	-	X	-	-	-	X	X	-	-	X	-	-	X	X	X
Team	-	-	-	-	-	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
CVS	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	X	X	X	X	X	X	X
CVS_Over_SSH	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	X	X	X	X	X	X
CVS_Over_SSH2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	X	X	X	X	X	X
SVN	-	-	-	-	-	X	X	X	X	X	X	X	X	X	X	X	-	X	X	X	X	X
Subversive	-	-	-	-	-	-	-	-	-	-	X	X	X	X	X	-	-	-	-	-	-	X
Subclipse	-	-	-	-	-	X	X	X	X	X	-	-	-	-	-	X	-	X	X	X	X	-
Subclipse_1_4_x	-	-	-	-	-	-	-	X	X	X	-	-	-	-	-	X	-	-	-	-	X	-
Subclipse_1_6_x	-	-	-	-	-	X	X	-	-	-	-	-	-	-	-	-	-	X	X	X	-	-
GIT	-	-	-	-	-	-	-	-	-	X	-	-	-	-	-	X	X	-	X	X	-	X
EclipseFileSystem	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Local	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Zip	-	-	-	X	X	-	-	-	X	-	-	-	-	X	-	-	-	-	-	X	-	X

Figure 3 Eclipse features covered by test suites - borrowed from [40]

2.5 Improving Variability Modeling

BVR – and its precursor CVL – have mainly been investigated in the ITEA2 MoSiS project[2], the FP7 VARIES project [41] and in previous DREAMS activities [2, 4]. BVR promotes variability as a separate concern and provides novel methods and tools to improve efficiency of the development process. Whilst modeling variability has now matured, less attention has been paid to realizations (see Section 2.1). We therefore have identified the following challenges:

1. **Decoupling variability and realization**. The structure of the feature tree (i.e., the variability specification) currently governs the behavior of the realization layer. The realization layer applies fragment substitutions [4] as it traverses the resolution tree. This implicit coupling between resolution and realization makes their definition tedious and error prone, and the resulting realizations brittle.
2. **Enabling "solid" realization layer**. To be effective, the realization of features must be commutative as they may be applied in any order. Engineering a commutative fragment substitution is difficult because testing is impractical with the current tooling, which does not support the execution of fragment substitutions in isolation.
3. **Ensuring technology independence**. The intent beyond BVR is too fully separate the variability issues, but the current tooling implies the use of modeling technologies because BVR was primarily meant to operate on MOF-compliant languages [42]. Ideally, variability should be defined regardless of any particular realization technology, whether they are deployable artifacts, modeling languages or something else.

[2] See https://itea3.org/project/mosis.html

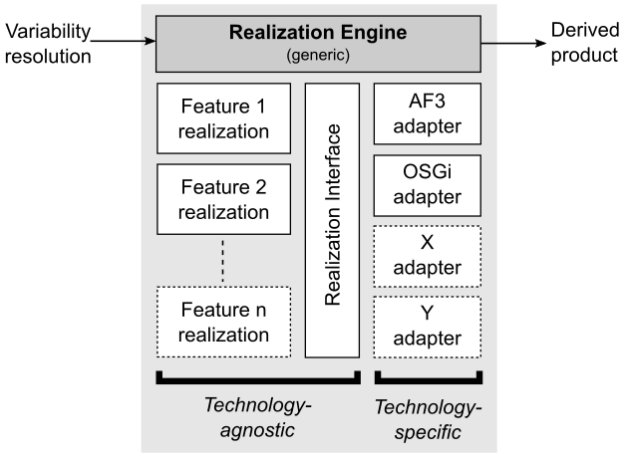

Figure 4 Toward an imperative realization engine for BVR

Our solution focuses on improving the realization engine of BVR, as shown on Figure 4. This new design is based on the following principles:

- **Shift from declarative to imperative realization**. Fragment substitutions were an attempt to provide a declarative way to model realization, by specifying pieces of models that must be substituted. We proposed to define realization as independent imperative fragments describing what must be done to enable a given feature (see "feature realization" on Figure 4). These fragments can then be tested in isolation.
- **Contract-based realizations**. Each imperative feature realization fragment is enhanced with a pre and a post condition. The order in which fragments are applied thus results only from the dynamic evaluation of pre-conditions (computed by the "realization engine" on Figure 4). In addition, the validity of the resulting product can be checked using post-conditions. Both pre and post conditions ease the diagnostic of issues in the feature realizations.
- **Technology-agnostic realization interface.** The imperative feature realization are built upon a technology-agnostic interface (see "realization interface" on Figure 4), which abstracts the technology specific details. Separate and yet interchangeable artifacts encapsulate implementation details specific to a given technology (called "adapter" on Figure 4). Both the technology-agnostic interface and its adapters become reusable between product lines within the same application domain.

3 Variability to Support Design Space Exploration

3.1 Overview

Design-space exploration (DSE) investigates – at design time – alternative solutions in order to obtain a balance between conflicting system properties such as functionality, cost, complexity or energy consumption to name a few. The proposed approach finds Pareto-optimal solutions based on the input goals configuration in DSE. This means that an optimization in one goal can lead to a quantitative decrease on other goal. For instance, a given solution could provide an optimal energy consumption using the selection of the corresponding platform architecture and optimized implementations of (logical) software components. However, this solution might not meet the safety qualification criteria for certification, and/or it could not comply with the defined real-time constraints, or its estimated engineering and production cost might exceed the defined budget. As pointed out in Section 1, ultimately choosing the best design remains simultaneously a *business* and a *technical* decision.

The DREAMS project therefore approaches design-space exploration at both the business and technical levels, using variability models and an evolutionary optimization, respectively (see D 4.1.2 [4], Section 3, for a comprehensive treatment). At the business level, the variability models capture the design decisions that govern what functionality will be offered by the system. At the technical level, the evolutionary DSE algorithm explores alternative engineering decisions resulting in contrasted trade-off between different extra-functional system properties. Although complementary, the final products yielded by both approaches must remain within the limits of the design problem.

Design problems are indeed constrained by the application or the application domain, which may require for instance compliance with laws or domain specific regulations and standards. In the context of MCS in particular, the compliance with safety standards (e.g., IEC 61508 [12-19], DO 178C [20, 43], ISO 26262 [22-31]) influences both the functional and extra-functional concerns. In general, to be pertinent and support the designer's decision process, design space exploration must be integrated with specific analysis (e.g., safety, temporal properties, etc.), to ensure that obtained solutions remain within the problem scope. For individual products (i.e., DREAMS system instances), such analyses have been presented in D 4.1.2 [4] where they are used to explore – in the terminology of this deliverable – technical design decisions. Section 3.5 will define analyses for the selected verification dimensions for mixed-criticality systems that also consider business decisions based on a given variability model defined at the technical level, the optimization algorithm explores alternative engineering decisions yielding contrasted trade-off between extra-functional concerns.

The process we propose in Figure 5 combines the exploration based on business variability specification with an evolutionary optimization based on a given set of goals (i.e., the defined design constraints and optimization objectives), while minimizing their overlap. In the following, we describe this combined process by splitting into several steps the common approach in product-line exploration (i.e., to derive from a product-line description a set of high-quality products defined as models in a given domain language).

- The description of a product line is a specification of the variability as well as all reusable assets from which final products can be constructed. Here, reusable assets are encoded as so-called *"150 %"-model* in the domain-language that is used to model the final products. In the proposed approach, the DREAMS application and platform-meta-model [2] is used as domain-language. Since this model contains all assets that can potentially be used in the final product, it is in general not a valid product model itself.
- As pointed out in Sections 2.3 and 0, the first step is to identify a set of resolutions that maximizes interaction coverage between the different features contained in the variability specification. Then, each resolution is realized by the BVR engine (see Section 2.5). Instead of directly constructing product models (as it would be the case in a traditional variability exploration process), the combined process yields a so-called "125 %" model.
- In the *"125%-model*", the variability has been only partially resolved. This partial variability resolution process focuses on the resolution of business decisions governing what features to include into a product. However, the model contains some remaining variability that mainly concerns technical decisions. This is because the resolution of these technical decisions requires information that is only available internally during the evaluation phase of the evolutionary optimization algorithm [4]. Here, a typical example is the execution schedule that is computed during the decoding step of the algorithm and that is required to perform decisions regarding the application of fault-tolerance mechanisms (e.g., whether to replicate components or to use diverse implementations).
- As pointed out before, most of the features are directly "realized" using some reusable assets in the first step. However, some specific features (e.g., those concerning safety), require further technical investigations and hence cannot be resolved using solely the variability specification. The evolutionary optimization engine collects the remaining technical variability decisions and searches among alternative implementations to obtain a complete product model (i.e., in the present approach to compute an application-to-platform

deployment and potentially a redundant version of the original application model [4]). Here, the exploration ensures that the selection of all extra-functional constraints (e.g., safety, timing) are met and that the solutions w.r.t. the defined objectives are optimized (e.g., energy consumption). The results form a Pareto-optimal set of contrasted *solutions*, i.e., valid *product* models – or "100 %" models.

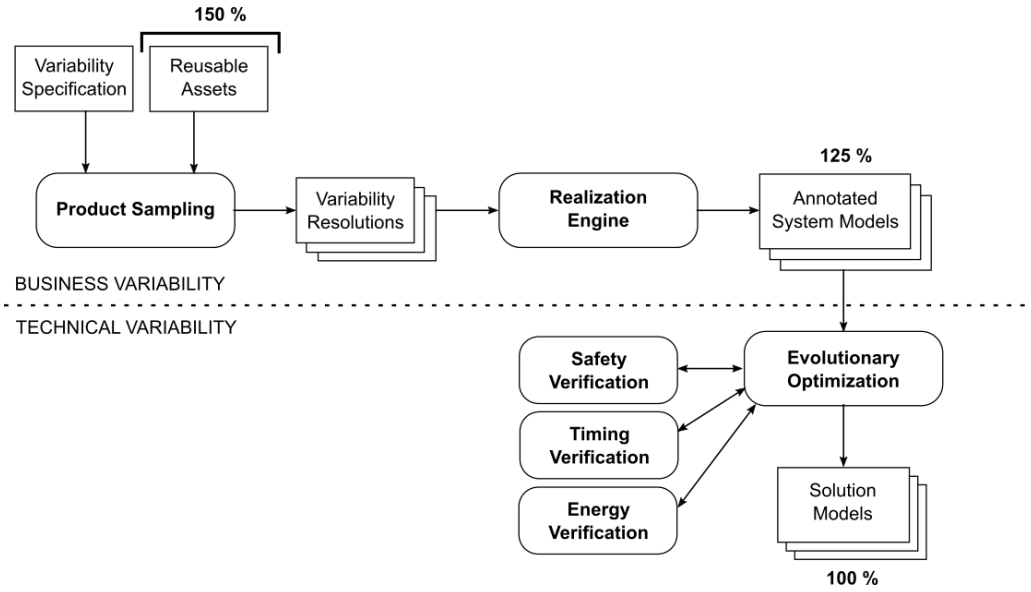

Figure 5 Integration of business and technical design space exploration

The DSE provides an optimization of the partially resolved models yielded by BVR, from which it generates a set of Pareto-optimal solutions (see Section 3.1.3 in D 4.1.2 [4]). For each of these solutions, the metrics computed during the evaluation phase of the evolutionary algorithm are provided. These metrics enable the designer to evaluate the quality of the selected "125 %"-models, and hence to select the best strategy for product sampling.

3.2 Variability Dimensions in Mixed Criticality Systems

Many elements and properties can affect critical aspects of safety, timing and extra-functional properties (i.e., reliability, energy consumption). It would be almost impossible and useless to list all the elements that can vary in DREAMS meta-models altering safety and timing. In the following, a list of *a priori* interesting variations is given (classified by meta-model) all of them heavily influencing safety and timing:

- *Component model* – the components of a system may vary in function of the features. Most interesting variability is:
  - Presence/absence of components
  - Management of logical ports connections of components
  - Redundancy of components
  - Different types of components (under replication or not)
  - Versions of different operating systems
  - Variations in the safety requirements (see example in Section 3.5.1.2)
  - Variable requirements properties
    - Force to a tile/core
    - Isolation in one partition
    - Access rights to hardware elements
- *Platform model* – The hardware elements of a system can vary. Most interesting variability is:
  - Presence/absence of hardware elements (nodes, tiles, cores, buses, watchdogs, clocks, RAM blocks, ROM blocks, etc.)

  - Management of bus connections when eliminated
  - Management of layout connections
  - Variations in the safety requirements (see example in Section 3.5.1.2)
  - Variations in the safety manual
- *System software model* – The hypervisors/partitions can vary. Most interesting variability is:
  - Presence/absence of partitions
  - Reallocation of hypervisors to other tile
  - Reallocation of partitions to other cores
  - Versions and different operating systems supported by hypervisors
  - Versions and different operating systems of the partition
  - Variation in the safety manual
- *Safety model* – All safety compliant items (SCI) corresponding to components, hardware elements, hypervisors and partitions can vary, as well as variations in the safety manual.
- *Timing model* – The timing model varies according to the selected software and software components.
- *Energy Consumption model* – the energy consumption model varies according to the selected software and hardware components.

The new design of the BVR realization engine (see Section 2.5) should support all these variations.

3.3 Variability Resolution & Realization

In Figure 5, we select a set of products, which are then fed as starting points for the evolutionary optimization. This selection includes *sampling* and *realization*.

During sampling, we select specific products. Yet, our selection strategy eventually influences how successful is the design-space exploration. Interaction coverage for instance is one strategy, which ensures that verification techniques are effective (see sections 2.3 and 0), but does not help with the relevance of the final products. Uniform sampling is an alternative where each product has the same odds to be selected. Uniform sampling does not help relevance either, but it is a baseline against which other strategies may be compared. Diversity sampling is known to fit well evolutionary optimization, because it yields products that are different from one another, and therefore avoids premature convergence [44]. In Task 4.3, we will investigate what sampling strategy, or combination thereof, best serve offline adaptation [4].

Then, during realization, we transformed these selected products into separate DREAMS system models. Sampling yields resolutions, which only describes the features included in each product. The realization of each product (see Section 2.5) implements the selected features, but if several implementations are possible, the final decision is delegated to the evolutionary optimization. In the DREAMS platform for instance, the deployment scheme does not alter the feature offered by the system, but affects the trade-off between extra-functional concerns. The optimization engine is therefore responsible for finding a relevant one.

3.4 Deployment Optimization

The core idea behind variability-based design space exploration is the mapping between the variability features described in the variability models, and (mainly technical) design decisions that steer the final product in a particular direction. As pointed out in D 4.1.2 [4], the DSE generates a set of solutions based on the given applications, platforms and goal specifications model. As sketched in Section 3.1, the focus of the DSE is the optimization of the system w.r.t. to "technical decisions" (e.g., such as deciding the best deployment of components to execution units), which in this deliverable will be combined with the aforementioned approaches to reason about "business decisions".

Section 3.1 introduced the notion of "125 %" models that are generated by the BVR engine based on the original “150 %”-input model (i.e., the reusable assets) and a variability specification. In this model,

variation points related to pure business decisions have already been resolved (e.g., whether to include an optional functionality into the system). Obviously, the variability already resolved by the BVR engine is transparent to the DSE and cannot be distinguished from a manually created input model. However, a "125 %" model can contain (partially) unresolved variability regarding technical decisions that can be optimized by the DSE. In the following, the encoding used in the "125 %"-model and the technical decisions considered by the DSE will be defined more precisely.

In Section 2.2, we distinguished between two different techniques for encoding variability in the model, namely annotation-based and composition-based approaches. On the one hand, the first one associates an architecture artifact with an annotation, which is closely related to variability features. On the other side, the composition-based approach relies on having a set of "composable" units. In the proposed approach, we support the variability at design space exploration level as a combination of both annotation-based and composition-based approaches to specify the variability features to be considered by the DSE – that have not been fully resolved by the BVR engine. In the following, both approaches will be detailed based on concrete technical decisions which will be supported by the DSE.

- To integrate support for a technical decision using *annotation-based variability*, a new annotation type is created using the DREAMS annotation meta-model [2]. Per definition, annotations are instantiated for all model element type they are registered for, and hence are already present in the initial "150 % model" (with default values). When creating the "150 % model" during the realization step, the BVR engine chooses based on the current resolution concrete values for the attributes of the annotation. This step is called "partial resolution" since – depending on the concrete choice for the respective features in a variability specification – different technical design-spaces might be encoded into the annotation modified by the resolution engine.

  In this task, the annotation-based approach will be used to consider redundancy in the variability exploration process. As pointed out in D 4.1.2 [4], the DSE applies redundancy in order to ensure that its solutions meet reliability goals. For this, it requires to know which logical components in a given architecture can be replicated. In the proposed approach, this information is provided using an annotation registered for logical components that allow for specifying a lower and upper bound for the number of replica. Based on business decisions, the BVR engine then sets concrete values for these bounds. The exact replication count and the applied replication scheme (spatial or temporal) that is used for a particular component is a technical decision that is hence performed by the DSE.
- The *compositional approach* means to have a diverse set of artifacts (e.g., logical components or platform elements) implementing the realization of one or several variation points. In the proposed approach, composition-based variability will be implemented using a so-called *artifact pool* containing all possible (technical) variation possibilities. Hence, the component-based approach can be used in order to encode the existence of different design variants of system elements.

  In the following, a use-case of this approach is presented that will be used to consider software design diversity in the DSE, i.e., to consider the existence of different implementations of a given component. A so-called *component pool* is used to encode in the "150 % model" all available variants of logical components. As shown in Figure 6, the component pool is modeled using dedicated logical architecture with the following semantics: Components on the first level are used to define the respective variation points, (i.e., the logical components for which design variants exist) including their input/output interface. The second level of the component architecture contains the different design variants of the respective component (including the annotation of its non-functional properties). These components must have the same interface as their root component at the first level of the component pool architecture. Additionally, the annotation-based approach is used to specify which variants exist for the logical components in the regular component architecture that defines the system's functionality. Here, the respective component in the regular component architecture is a

placeholder that contains an annotation with references to all available variants in the component pool (the placeholder component must have the same interface as its variants). Now, the same approach as discussed above can be applied, i.e., based on business decisions, the BVR engine can set concrete values for the references in the annotation (defining the set of available variants). Picking a concrete variant is a technical decision again and is therefore performed by the DSE.

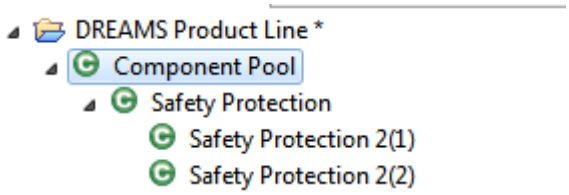

Figure 6. Example of a Component Pool

In the previous paragraphs, it has been discussed how the remaining variability is represented in "125 % models", and how these models can be generated. In the following, we discuss the how the DSE and processes these models in order to obtain "100 % models" (i.e., products, or solutions). While the encoding used by the evolutionary optimization (see D 4.1.2 [4], Section 3.1.4) is able to represent the spatial and temporal replication of logical components, it is not capable of directly considering software design diversity. Hence, in the course of Task 4.3, either the encoding will be extended accordingly, or a transformation will be defined that maps the corresponding information contained in the "125 % model" to the original encoding.

As mentioned in the beginning of this section, one part of the input model to the DSE (i.e., "125 % model") is a goal specification model that defines which optimization criteria should be considered during the exploration. The output of the DSE is the set of Pareto-optimal solutions according to the defined goals. Each of the solutions is rated with a metric according to the defined goal specification model. These metrics and their use as verification dimensions for mixed-criticality systems will be discussed in the next section.

3.5 Verification Dimensions in Mixed-Criticality Systems

Verification dimensions serve as a qualitative and quantitative evaluation of the output models produced by the DSE. The qualitative dimensions can be realized as constraints over the deployment model (e.g., safety constraints) and quantitative dimensions as optimization objectives to maximize the different optimization goals (e.g., reliability, end-to-end timing or energy consumption). The DSE derives these output metrics from the optimization goals defined as part of its input model. The metrics are based on parameters from both the logical architecture and the platform architecture, and are highly dependent on a concrete deployment using in a particular solution produced by the DSE.

In the overall variability exploration process (see Section 3.1), the goal specification is provided in the "150 % model" and – after the resolution of the business decisions – passed to the 125 % model.

In this section, we present a definition of the most relevant verification dimensions for mixed-criticality systems. While qualitative verification dimensions ensure that only valid output system configurations are generated under the consideration of variability, the metrics associated to quantitative dimensions enable the designer to compare and select from the generated set of products.

3.5.1 Safety

The verifications that can be performed using the safety model can be classified into three groups, as follows:

- Safety allocation constraints to verify a deployment allocation;

- Safety rules to verify safety properties of the emerging system;
- Safety rules to verify DREAMS safety cases of hypervisors/partitions/tiles.

In this context, it is important to note that "*verified product*" means that, *with the information available in the safety model*, no errors can be detected.

Theoretically, for every product, all three verifications must be performed to consider a product *verified* from the safety point of view. The key idea is to use Combinatorial Interaction Testing (see Section 2.4) to verify just a subset of products and to *save time* compared with a systematic verification of all products.

The next three subsections describe these constraints and rules. In the next Deliverable 4.3.2, an exhaustive list of constraints and rules will be implemented. Finally, we show how the verification dimensions relate to the certification activities carried out in Task 5.5.

See *Appendix A* for documented examples of Safety Allocation Verification Constraints and *Appendix B* for documented examples of Safety Verification Rules.

3.5.1.1 Safety Allocation Constraints to Verify a Deployment Allocation

Definition

A *safety allocation constraint* can be defined as a condition (that represents allowance or not allowance of a set of mappings) that a *deployment* has to satisfy. Otherwise the deployment conflicts with the IEC 61508 [12-19] safety standard. If this condition is not met, the deployment is considered invalid (in the sense that certification is not possible) and is discarded.

Example

If a software partition P1 claims to be SIL 2 and a software component C1 requires SIL 3, C1 cannot be deployed in P1.

Scope

A deployment theoretically allocates components to partitions, partitions to hypervisors, hypervisors to tiles and partitions to cores (of the tiles to which its hypervisor is mapped). The following group of constraints (detailed in Appendix A) will be generated and checked for every deployment:

- Safety Constraints for components mappings
  - Safety components only into safety partitions
  - Non safety components only into non safety partitions
  - Safety components isolated in one partition *(when required)*
  - Safety components specifying tiles/cores *(when required)*
  - SIL claimed by component supported by partition
  - All components in a partition have the same SIL
- Safety Constraints for partitions mappings
  - Safety partitions only into safety hypervisors
  - Safety partitions only into safety cores
  - Safety partitions specifying cores *(when required)*
  - SIL claimed by partition supported by hypervisor
- Safety Constraints for hypervisors mappings
  - Safety hypervisor only into safety tiles
  - Safety hypervisors specifying tiles *(when required)*
  - SIL claimed by hypervisor supported by tile

Usage and Description

Testing safety allocation constraints allows the DSE tool to rule out unsafe deployments. Before beginning its exploration, the DSE calls the safety module, which produces the safety allocation constraints that any valid deployments must satisfy. Whenever the DSE finds a new deployment, it calls the safety module to verify if these constraints are met, discarding the deployment otherwise.

Figure 7 below shows some of these constraints produced by the safety tool for the whole product-line.

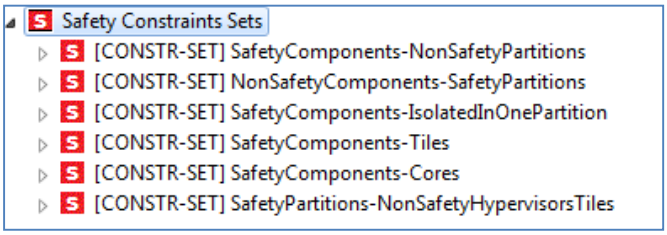

Figure 7. Safety Constraint Sets

Note that safety constraints are generated before the business variability gets resolved but are evaluated when a deployment is generated. However, the safety rules (see Sections 3.5.1.2 and 3.5.1.3) are always evaluated after the technical variability is resolved, when a complete system is generated.

3.5.1.2 Safety Rules to Verify Safety Properties for the Emerging System

Definition

We define a safety rule as a set of conditions that represents requirements from the IEC 61508 [12-19] standard. If these conditions are not met, the system is considered unsafe (in the sense that certification is not possible). Safety rules cannot be expressed as static constraints.

Example

As a simple example of safety rule, consider a hardware platform tile T1, which contains variability that leads to different SIL levels. We cannot define a constraint that a priori says which hypervisors can be mapped to that Tile because the SIL level of the tile is only known *a posteriori*. Therefore, this verification has to be implemented as a safety rule that is evaluated when the system is completely defined.

Another example is a hardware component that, depending on some features, may contain one or two independent cores. When redundant cores are present (and provided that requirements for on-chip redundancy detailed in IEC 61508-2 Annex E are met), a theoretical hardware fault tolerance (HFT) of 1 can be reached. With only one core however, an HFT of only 0 can be justified.

- The HFT level changes the support of SIL claims. For example, in case a SIL 3 level is required (as in protection function of the wind power use case), according to Table 3 of IEC 61508 2: the SIL 3 level with a HFT of 0 requires a high diagnostic coverage (DC ≥ 99 %)
- SIL 3 level with a HFT of 1 requires a medium diagnostic coverage (90 % ≤ DC < 99 %). It is difficult and costly to reach a high DC. Therefore a strategy is to develop elements with “just” a medium DC and use redundancy to achieve and HFT of 1, therefore, resulting in a theoretically possible SIL 3 level.

This kind of safety rules can impact dramatically in the DSE search procedure. For example, consider that the user is allowed to choose the SIL levels she wants for the safety protection function. In this case:

- If the user requires SIL 2, then the DSE may find a solution with only one core and only one safety Protection + Diagnostic components.
- If the user requires SIL 3, the DSE shall find a solution with two cores and shall replicate both the safety protection and the diagnostic components.

Therefore, variability in safety dimension has a high impact in the type of solutions.

Scope

Safety rules affects basically to safety compliant items for which a safety manual has been defined. 0 shows some examples of the safety rules to be implemented in Deliverable 4.3.2.

Usage and Description

The SIL level of a system is an emerging property that depends on multiple factors including for instance the safety SIL levels of components/hardware/system software, how integration has been carried out or whether safety manuals have been followed.

As mentioned before, there are safety verifications for which it is not possible to create, a priori, a constraint defining what *can be done* and what *cannot be done*. Such verifications have to be carried out when the product is completely defined.

This category of safety verification is implemented by a set of safety rules that are evaluated for each product generated but, as in the case before, product-line sampling techniques will be used to limit the number of products for which these verifications have to be done.

3.5.1.3 Safety Rules to Verify DREAMS Safety Cases of Hypervisors/Partitions/Tiles

A special subset of safety rules can be implemented to verify if some conditions required by safety cases developed in Task 5.1 are satisfied.

The future Deliverable 4.3.2 will identify and implement some verification rules corresponding to safety cases of hypervisors / partitions / tiles.

3.5.1.3.1 Rules to Help Verifying Safety Cases for Hypervisors/Partitions

Task 5.1 has produced a modular safety case (see Deliverable 5.1.1 [45]). It defines a set of minimum reasonable arguments and evidences that a hypervisor/safety software partition should meet/provide in order to claim IEC 61508 [12-19] compliance and support the reusability of hypervisor/partition (e.g., in a product-line). It is a top level safety case for a hypervisor/partition. Some of these arguments can be verified on the meta-models by implementing safety rules. This set of minimum reasonable arguments considers that the partition provides / defines:

- Safety techniques and measures to reduce the probability of systematic faults, developed for instance, in compliance with IEC 61508 [12-19].
- Safety techniques and measures to control errors: Required diagnosis techniques and required system reaction to errors.
- Safety related functions: Safe start-up and initialization and safe shutdown.
- Safety related constraints and hypotheses: Constraints and hypotheses.

As stated in the D 5.1.1 [45], the set of minimum reasonable arguments consider that the hypervisor defines:

- Safety techniques and measures to reduce the probability of systematic faults (e.g., developed in compliance with IEC 61508 [12-19]).
- Safety techniques and measures to control errors: "Required diagnosis techniques [...]" and "Required system reaction to errors [...]"
- Safety related functions: Virtualization of resources, exclusive access to peripherals, start-up and initialization, shutdown, configuration, time independence, spatial independence, etc.
- Safety related constraints and hypotheses: Constraints and hypotheses.

Some of these *minimum* can be verified by implementing the corresponding safety rules.

3.5.1.3.2 Rules to Help Verifying Safety Cases for Tiles

In the same way, Task 5.1 defines in Deliverable 5.1.2 [46] a modular safety case for COTS processors. This safety case defines a set of minimum and reasonable arguments and evidences that a COTS multicore device should meet / provide in order to enable / support the development of MCS compliant with IEC 61508 [12-19].

As stated in Deliverable 5.1.2 [46], the minimum reasonable set of safety arguments that a safety device must provide are:

- Safety assumptions: Constraints and hypothesis.
- Safety standards for the management and reduction of systematic faults: Compliance with IEC 61508 [12-19] safety standard and qualified tools.

- Safety techniques for the reduction, management or avoidance (as much as possible) of interference among elements defined in IEC 61508 Annex F (e.g., safe power up and boot, safe shutdown and power down, configuration).
- Safety techniques for the reduction, management or avoidance of faults (as much as possible). Safety techniques include diagnosis techniques, system reaction to errors and fault tolerance techniques.
- Recommended usage of resources and peripherals.

Regarding *diagnosis techniques* for example, Figure 8 below (borrowed from [46]) shows part of the "*Diagnosis Techniques*" recommended (*dark grey*) for a ZynQ device (used in the wind power demonstrator) to achieve a SIL 3 certification.

	Diagnosis Technique/measure	See IEC 61508-7	Maximum achievable DC	Supported by ZYNQ	Argumentation
Processing Unit (CPU)	Comparator	A.1.3	High (99%)	No	
	Majority voter	A.1.4	High (99%)	No	
	Self-test by software (limited number of patterns)	A.3.1	Low (60%)	No	
	Self-test by software: walking bit	A.3.2	Medium (90%)	No	It can be supported by software implementation
	Self-test supported by hardware	A.3.3	Medium (90%)	No	
	Code processing	A.3.4	High (99%)	No	It can be supported by software implementation
	Reciprocal comparison by software	A.3.5	High (99%)	No	It can be supported by software implementation.
Invariable Memory (ROM OCM)	Word-protection multi bit redundancy	A.4.1	Medium (90%)	No	
	Modified checksum	A.4.2	Low (60%)	Yes	It is implemented at boot time to verify the boot header.
	Signature of one word (8-bit)	A.4.3	Medium (90%)	No	
	Signature of a double word (16-bit)	A.4.4	High (99%)	No	It is supplied by software implementation
	Block replication	A.4.5	High (99%)	No	It is supplied by software implementation
Variable Memory (SDRAM, OCM)	Ram Test "checkerboard" or "march"	A.5.1	Low (60%)	No	It is supplied by software implementation
	RAM Test "walkpath"	A.5.2	Medium (90%)	No	
	RAM test "galpat" or "transparent galpat"	A.5.3	High (99%)	No	
	RAM test "Abraham"	A.5.4	High (99%)	No	
	Parity-bit for RAM	A.5.5	Low (60%)	Yes	It is supported by both L1 and L2 caches and OCM.
	RAM monitoring with a modified Hamming code or detection for data failure with error. detection-.correction codes (EDC)	A.5.6	Medium (90%)	Yes	ECC: It is supported by the DDR memory controller, QSPI, NAND flash CRC: It is supported by SD/SDIO host controller, PL and OCM.
	Double RAM with hardware or software comparison and read/write test	A.5.7	High (99%)	No	

Figure 8. Part of IEC 61508 compliant Random Failure diagnosis techniques for the ZYNQ device [12-19]

In this table, for each *diagnostic technique*, the third column represents the *maximum achievable diagnostic coverage (DC)*, the fourth column states whether ZynQ supports this *diagnostic technique*, and the last column represents how to add support for that *diagnostic technique* (in case the ZynQ does not support it).

A clear example of safety rule that can be implemented corresponds to the A.9.2 technique (shown in Figure 9 below). This technique can reach a 90 % DC that, coupled with a HFT of 1, could justify a SIL 3 level. Unfortunately, ZynQ does not provide a hardware watchdog (only a software one, see Column 4) and therefore an external watchdog will be needed (see Column 5). The existence of such a watchdog can be verified by a safety rule (see rule 8.1.1 in Appendix B).

Temporal and Logical program sequence monitoring	Watchdog with separate time base without time-window	A.9.1	Low (60 %)	No	
	Watchdog with separate time base and time-window	A.9.2	Medium (90%)	No	The device provides a software watchdog. Hence, an external watchdog is required.
	Logical monitoring of program sequence	A.9.3	Medium (90%)	No	
	Temporal and logical monitoring	A.9.4	High (99%)	No	It is supported by an external watchdog
	Temporal monitoring with on-line check	A.9.5	Medium (90%)	No	It is supported by an external watchdog
Clock	Watchdog with separate time base without time-window	A.9.1	Low (60 %)	No	
	Watchdog with separate time base and time-window	A.9.2	High (99%)	No	The device provides a software watchdog. Hence, an external watchdog is required.

Figure 9. External Watchdog recommended for ZynQ

3.5.1.4 Connection of Verification Dimension of WP4 T4.3 to WP5 T5.5

3.5.1.4.1 Generation of Verification Trace Evidences

Task 5.5 aims at developing a certification methodology for MCS. This methodology will be based on the use of cross-domain mixed-criticality design patterns and on the safety concepts of hypervisor, partition, tiles, etc. It will define the set of arguments and evidences that must be provided with a system in order to claim compliance with IEC 61508 [12-19].

Some of these evidences may be supported by documented traces of the safety compliance verification checks made during design space exploration. Such traces are automatically produced and support the certification methodology defined in Task 5.5.

3.5.1.4.2 Verification of documental evidences

Apart from verifying measurable safety properties, the safety manual could be enriched to contain pointers to the documents required by the certification methodology. At least, the presence of such documents can be tested, warning the user if any of them is missing.

3.5.2 End-to-End Timing

Meeting end-to-end deadlines is essential for MCS [10], since a missed deadline can harm the correct execution of a safety critical function. End-to-end deadlines are hard constraints to the final products, which must be kept independently of the decisions.

As pointed out in Section 3.1.7.2 of D 4.1.2 [4], the DSE verifies in its evaluation stage end-to-end deadline constraints in order check if a component-to-execution unit mappings is feasible. For this, the DSE decodes a given mapping into a time-triggered schedule that considers both computational and communication resources to provide an estimation of the latencies introduced by the distributed execution of software components. For each defined end-to-end deadline, the latency between the *start* and the *end* component is analyzed and compared to the defined deadline. The satisfaction of end-to-end constraints is indicated by the following metric: If the latency matches the deadline, a value of zero is returned. Otherwise the return value is the squared mismatch between the latency and the deadline (unit: seconds^2).

This metric is used in the variability exploration process as follows: Decisions at the business level influence which latencies are achievable in a system (e.g., because of the resource demands of the resulting application, and/or the resources provided by resulting platform). Hence, the decisions at the business level constrain the solution space for the DSE at the technical level. In case a product configuration resulting from a given set of decisions at the business level is infeasible according the defined end-to-end timing constraints, the set of solutions returned by the DSE is empty. Otherwise, the end-to-end timing metric guides the evolutionary optimization algorithm implemented by the DSE to determine a component-to-execution unit mapping satisfying all defined end deadlines.

Consider as an example a simple end-to-end constraint between the source and the sink component in a chain of components. Here, the worst-case execution time (WCET) of a component C_A of that chain of components varies both depending on the selected implementation (design diversity) and the timing of its execution unit. Thus, the choice to allocate a component C_A^X with a large WCET can lead to a violation of a timing constraint, whereas the allocation of a component C_A^Y with a smaller WCET on the same execution unit would not violate this constraint. As pointed out in Section 3.4, the set of available variants for a given component is influenced by decisions taken at the business level whereas the choice of a concrete variant from this set and the selection of its execution unit are technical decisions.

3.5.3 Reliability

Reliability is defined as the probability that a system provides its correct service in a given time interval [47]. For mixed-criticality systems, this metric is, on the one hand, relevant to harden critical parts of the system, i.e., to increase the reliability by applying fault-tolerance techniques that minimize the expected occurrence of service failures. On the other hand, an improved reliability will typically

increase the expected maintenance time intervals (there may also be other – application (domain) specific – factors).

As pointed out in Section 3.1.7.4 of D 4.1.2 [4], the reliability is computed for a specific deployment of a given application to the execution platform. It is determined using a binary tree analysis that evaluates the time-triggered schedule calculated in the decoding phase of the evolutionary optimization algorithm. For a given component, the failure probability is calculated based on its annotated WCET, the annotated failure rate of the execution unit to which it is mapped in the current deployment. The analysis starts with the estimation of fault/failure probability of a sink component (i.e., a component without output ports). Then it recursively considers the failure probability of the causal predecessor components (as defined by the data flow in the component architecture) of the currently invested component.

More precisely, the reliability metric is defined for a given application, i.e., the sub-graph of the logical architecture that is reachable form the given sink component as the failure probability in FIT (failures per our) within one period of the hyper-period of the calculated time-triggered schedule.

During the optimization, the DSE tries to increase the reliability of the system by introducing redundancy (i.e., temporally or spatially replicating components and adding voting components to the logical architecture) as well as preferring to allocate critical components to execution units with a lower annotated failure rate (only relevant for heterogeneous architectures).

3.5.4 Energy Consumption

The energy consumption can only be estimated for deployed systems, i.e. for a concrete mapping of logical components to execution units, and a corresponding execution schedule. The analysis used in the evaluation phase of the DSE is based on a linear power model that considers both the execution of components on computational resources (based on the estimated WCET) and the energy consumed by the communication of components via the platform (see Section 3.1.7.3 in D 4.1.2 [4]). The analysis sums the energy consumption of each resource in the platform in order to obtain a metric for the energy consumption of the whole system and thus enable distinguishing products by their energy demand. First, it calculates the energy consumption for a single hyper-period of the time-triggered schedule. Then, in order to provide a metric for the energy consumption of a deployment, it derives the average energy consumption by dividing the energy consumption by the length of the hyper-period.

Energy consumption is a relevant metric for mixed-criticality systems since its minimization conflicts with the goal to reduce the failures in the system based on replication. Furthermore, while deploying logical components to computing cores with a lower operating frequency can significantly reduce the energy consumption; such choice could conflict with the temporal requirements that are typically present in MCS.

In the context of product lines, the set of features of a specific product influence the energy consumption to a large degree. For instance, an increase in features can potentially increase the energy consumption of a system. Hence, this metric adds an interesting aspect to the choice of concrete products lines as the energy consumption has a large impact on the product as a whole, e.g., the dimensioning of batteries and the running costs.

However, not only the selection of features at the business level has an impact on the resulting energy consumption. As pointed out in Section 3.4, a component pool can be used to specify that different implementations of a logical component are available. Hence, the DSE can take into account the respective energy consumption of the different variants of a given component. The DSE needs to balance the decision regarding the energy consumption with the other relevant verification dimensions (for which the respective properties of the variants of the logical component might differ significantly).

4 Example: Wind Power Safety

4.1 Modeling the Variability of Safety Components

Safety protection components are required for safety critical systems. Yet, different levels of safety may be provided, depending on whether safety components are replicated, and whether replicas are all similar. Figure 10 illustrates one possible model for this variability – other organizations of features/choices remain possible. At the top level, a safety protection mechanism can be either replicated or simple. If the safety protection mechanism is replicated, its replicas may be diverse (e.g., resulting from N-version programming) or homogeneous (if they just are instances of the same code).

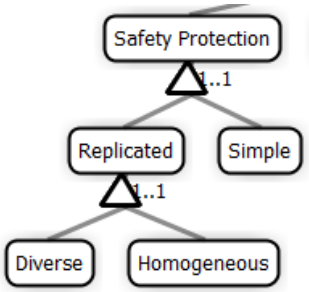

Figure 10 BVR models of variability in safety protection mechanisms

As mentioned before, variability in the wind power use case can be found in components, platform, system software and safety model. In this section variability resolution of the safety protection and diagnostic functions and supervision function is described.

4.2 Variability Resolution

4.2.1 Safety Protection and Diagnostic Functions

The safety protection function is in charge of maintaining the wind turbine in a safe state. The main functionality of the protection system is to assure that the design limits of the wind turbine are not exceeded.

The safety protection function may be *redundant* to achieve higher SIL level. In addition, under redundancy, both safety protection applications can be an exact replica (same software code) or can be *diverse*: different software code. In Figure 11 below, the software components of the wind power demonstrator are shown, where SafetyProtectionARM-2(1) is an exact replica of SafetyProtectionMicroBlaze and SafetyProtectionARM-2(2) is a different implementation of the Safety Protection function.

In addition, the diagnostic function must also be redundant when the safety protection is redundant, although in this case, no diversity is applied to the diagnostic function.

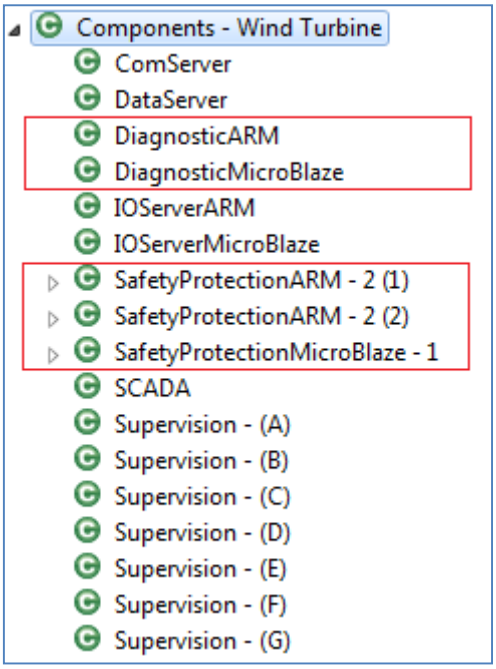

Figure 11 Software components in the wind power use-case

At the system software level, each of these components execute within their own safety partition, more precisely:

- Safety ProtectionMicroBlaze and DiagnosticMicroBlaze run both into FPGA Microblaze but in different partitions.
- Safety ProtectionARM and DiagnosticARM run both into ARM Cortex A9 but in different partitions.

Figure 12 below shows one possible system software configurations (hypervisors/partitions) for the wind power use case.

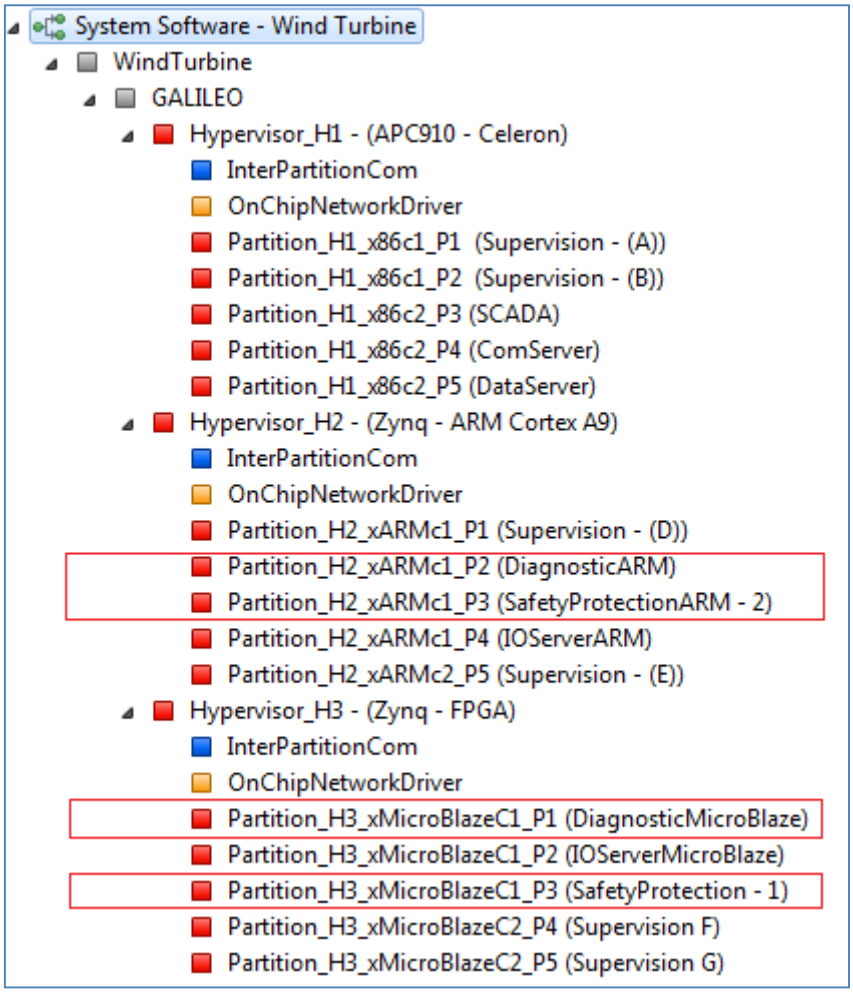

Figure 12. Partitions and Hypervisors of the wind power use-case

Finally, at the safety model level, the software components have their own requirements, and the hypervisors, partitions, tiles and nodes have their own safety manual. Figure 13 shows the safety model entities involved in the safety protection function.

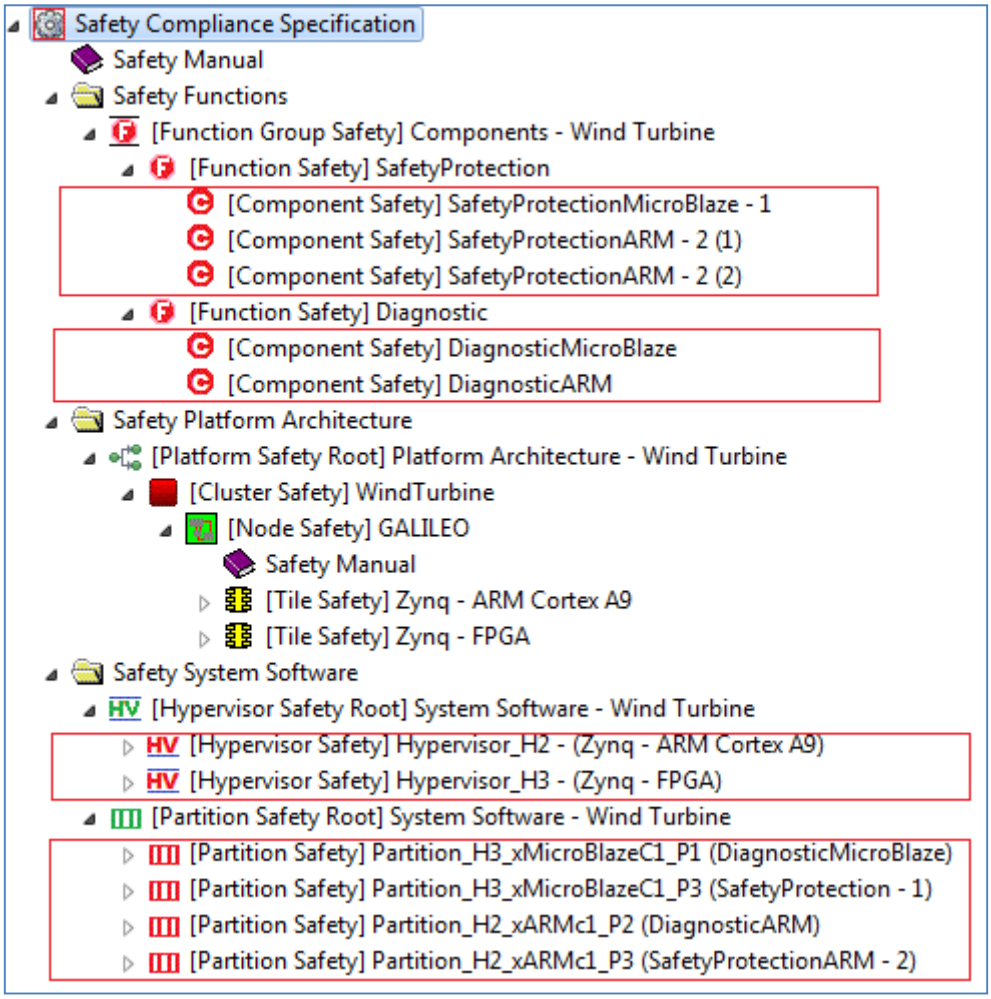

Figure 13. WP7 Wind Power Safety Protection function related entities in Safety Model

In the variability resolution phase, the user has to choose *redundancy* and/or *diversity*. This will yield three valid products, described in the following sections.

4.2.1.1 Safety Protection, without Redundancy

In this case, only one copy of the safety protection would be running in the MicroBlaze processor. Then, according to Table 3 in IEC 61508-2, with a HFT of 0 and a high diagnostic coverage (90 % ≤ DC < 99 %), only the SIL 2 level could theoretically be reached. The safety deployment would be valid only if other hardware elements and their corresponding diagnostic would increase the DC to 99 % (this can only be checked by a human expert). The safety verification would emit a warning, which may prevent an approval by a safety expert.

Figure 14 below shows a wind power product where the safety protection is not redundant.

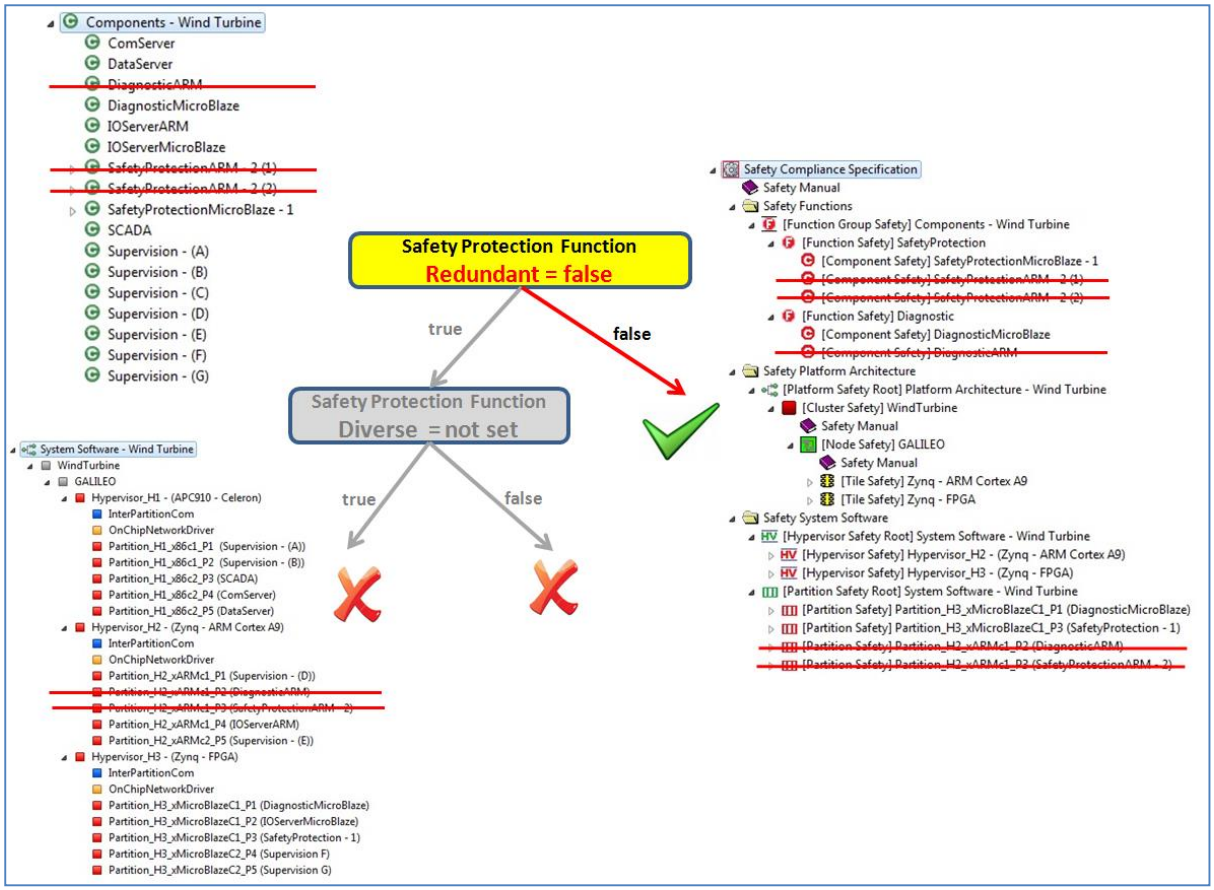

Figure 14. Safety Protection function NOT redundant – impact in models

Note that, in addition to these models, these features would also impact other models such as timing model, power consumption model, logical input/output ports connections of components, etc.

4.2.1.2 Safety Protection with Redundancy but without Diversity

In this case, two exact copies of the safety protection are running in two (arguably) independent hardware elements (ARM and FPGA) achieving a HFT of 1. As the components themselves have a DC between 90 % and 99 % then, with a HFT of 1, a SIL 3 level can theoretically be reached.

Figure 15 below shows a wind power product, with a redundant and homogeneous safety protection.

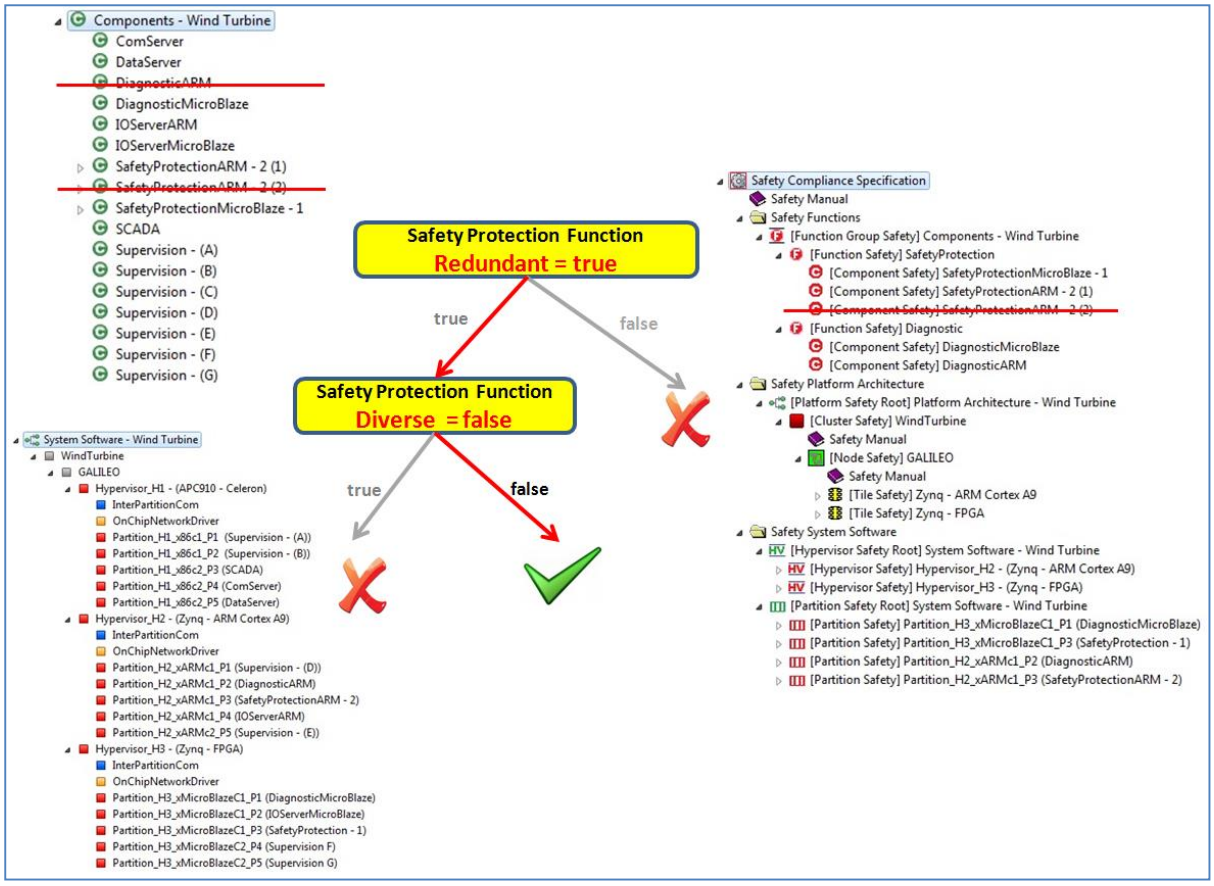

Figure 15. Safety Protection function redundant but NOT diverse – impact in models

4.2.1.3 Redundant and Diverse Safety Protection

In this case, two different copies of the safety protection are running in two (arguably) independent hardware elements (ARM and FPGA) achieving a HFT of 1. As the components themselves have a DC between 90 % and 99 %, then with a HFT of 1, theoretically, a SIL3 level can be reached.

Figure 16 below shows a wind power product with a redundant and diverse safety protection.

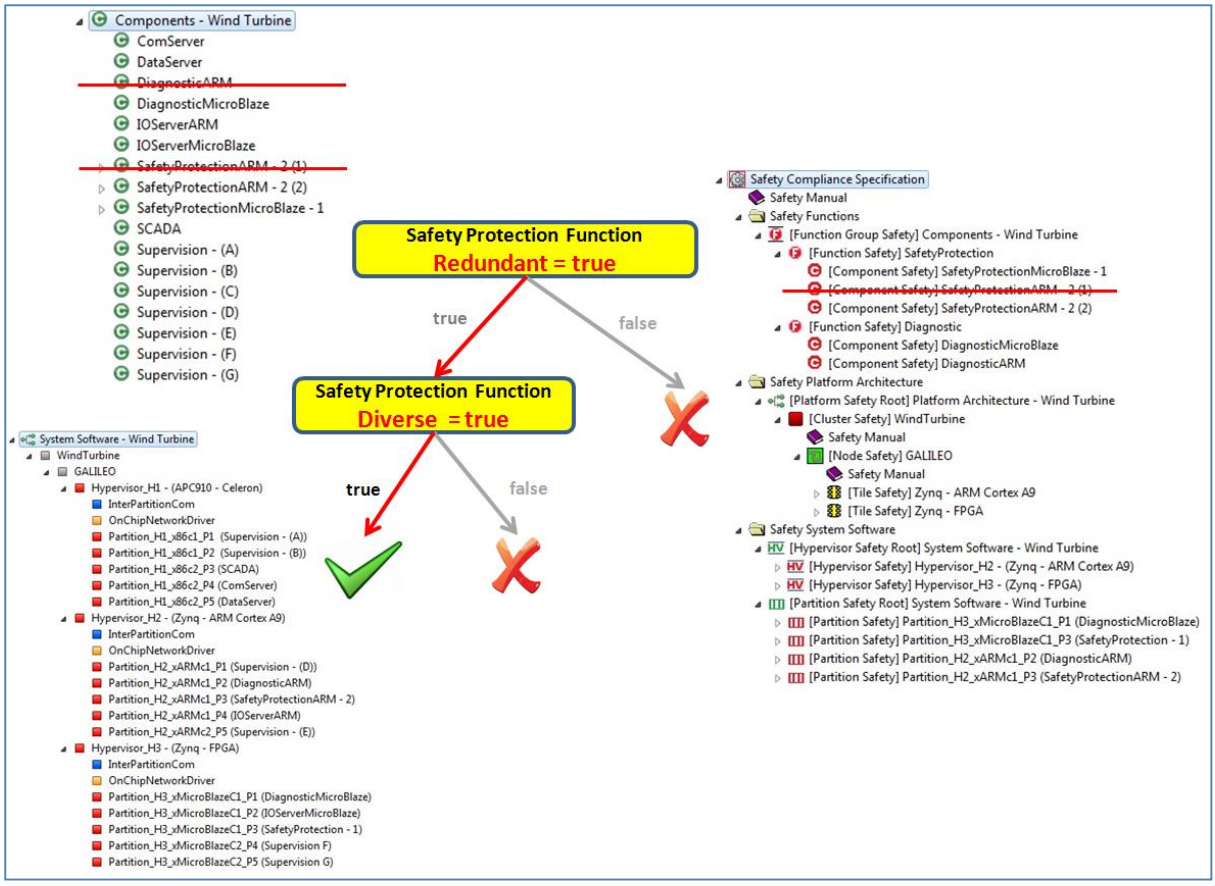

Figure 16. Safety Protection function redundant and diverse – impact in models

4.2.2 Speed, Vibration and Voltage Sensors in Safety Protection

The safety protection function processes information including speed, vibration and voltage to ensure that the design limits of the wind turbine are not exceeded, when facing strong winds for instance.

Inputs from physical sensors enter into wind power safetyrelated system via the EtherCAT bus. These physical sensors can be redundant, diverse (different model, version, etc.), or both. This increases further variability within the wind power demonstrator.

Although physical sensors are not modelled in AF3, the implementation of the SafetyProtection component is not the same. There are two alternative implementations for the safety protection:

1. The number of sensors is discovered dynamically during start-up
2. For each combination of sensors, there exists one different compiled version of the safety protection for that very number and type of sensor (this last option is easier to certify)

Although this is not decided yet, (will be decided in WP7) this variability exists and affects to logical components to be selected for a deployment. Depending on these features, one or another type of component has to be included in the final deployment. Most probably, only the SafetyProtection component corresponding to two sensors will be implemented.

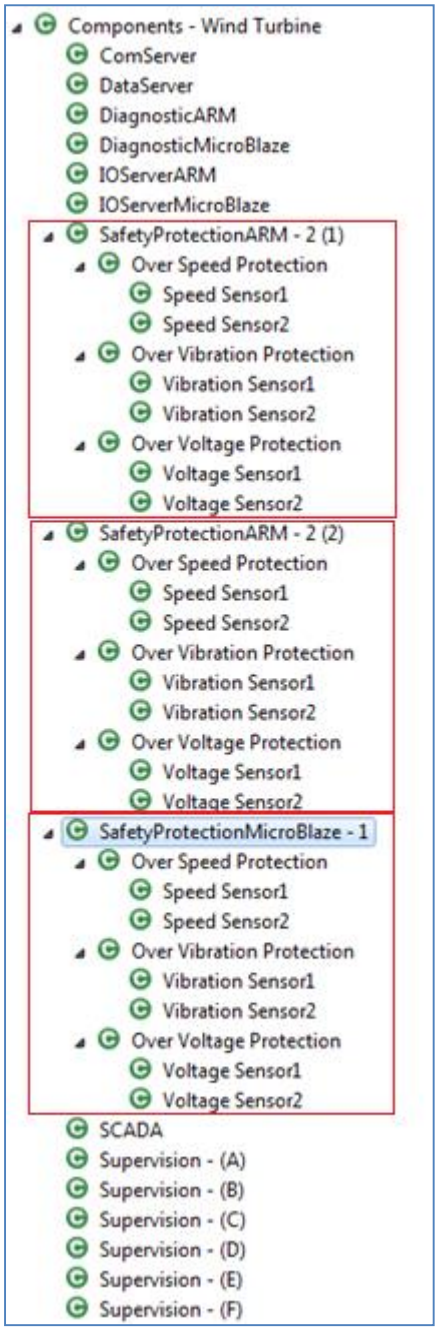

Figure 17. WP7 Wind Power variability in speed, vibration and voltage sensors

In the variability resolution phase, the user chooses **redundancy** and/or **diversity**, as shown in Figure 18 below.

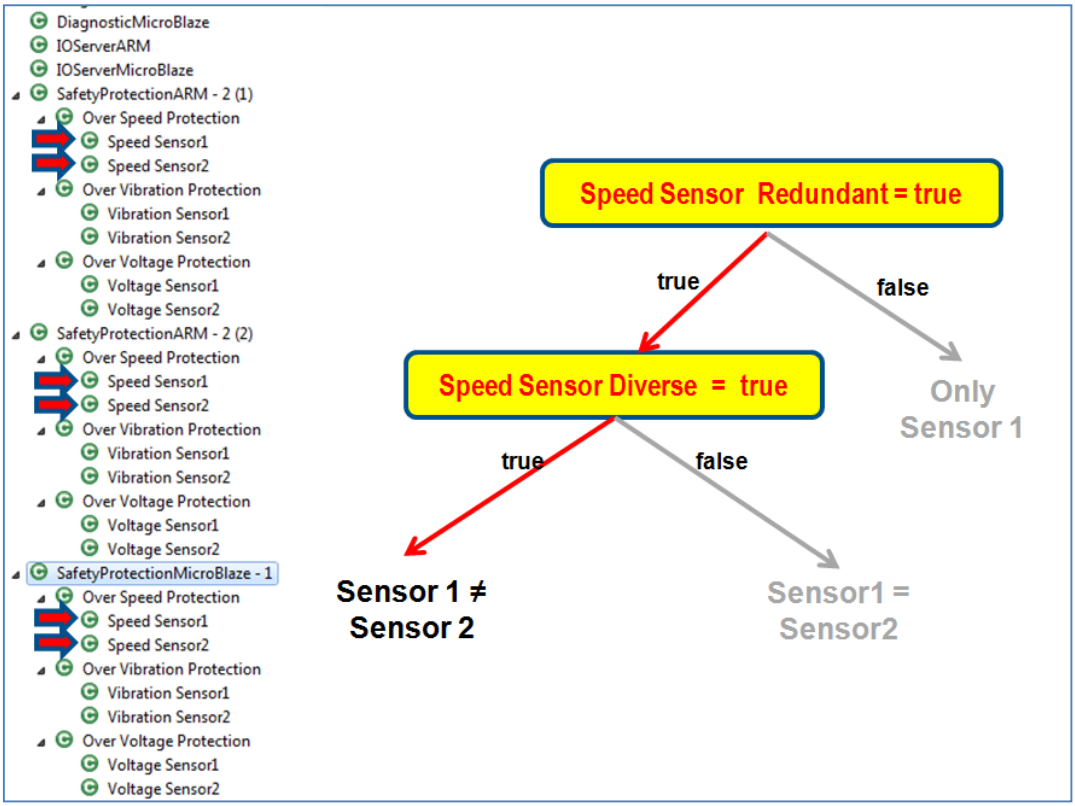

Figure 18. WP7 Wind Power with two sensors

4.2.3 Supervision Function

The supervision (and control) software component of the wind turbine includes a variable number of other components. Figure 19 below shows supervision components.

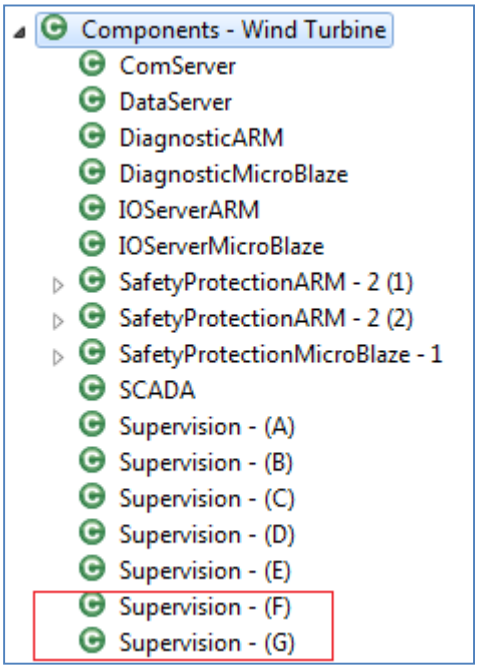

Figure 19. Extra Supervision components of WP7 wind power use case

Supervision Components A to E are always present in the wind turbine. However, the user may want to add two extra supervision components (F and G) by selecting feature *SupervisionPlus*. These two extra components would be allocated in two extra partitions as shown in Figure 20 below:

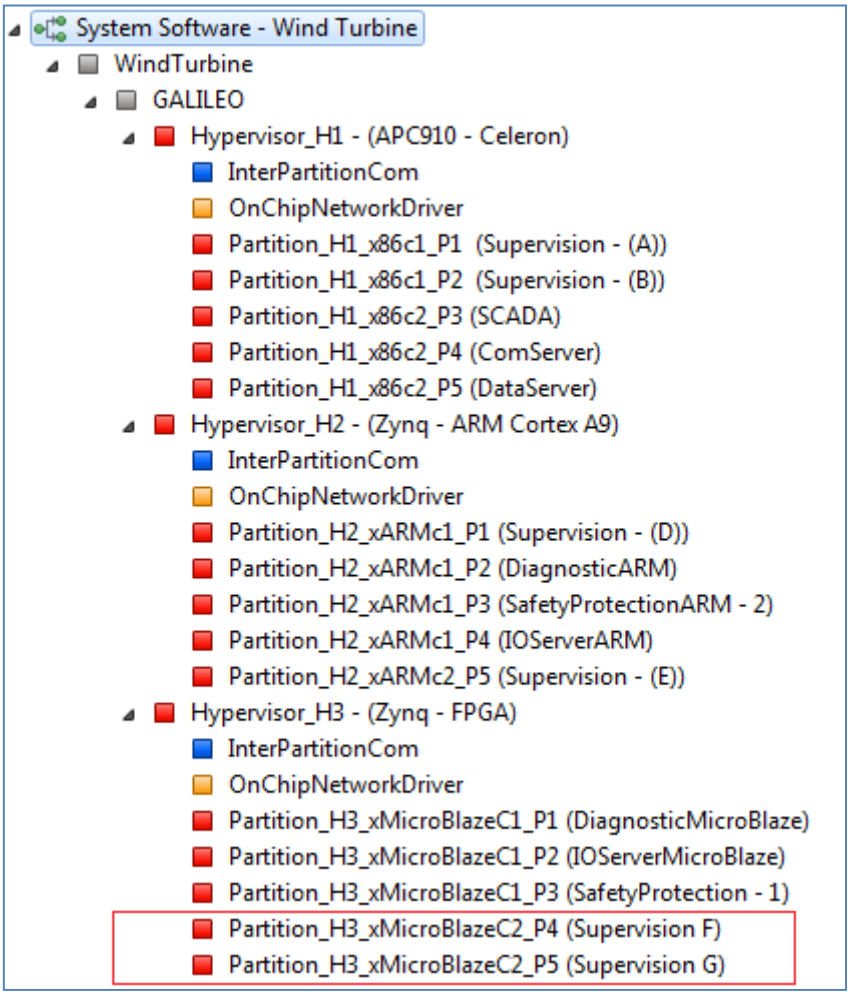

Figure 20. Additional partitions for Supervision extra components F and G.

Although it is not a current case, it is proposed that when these two extra partitions are created for the Supervision additional components, then two extra cores will be added to the FPGA MicroBlaze (with their corresponding RAM), as shown in Figure 21 below.

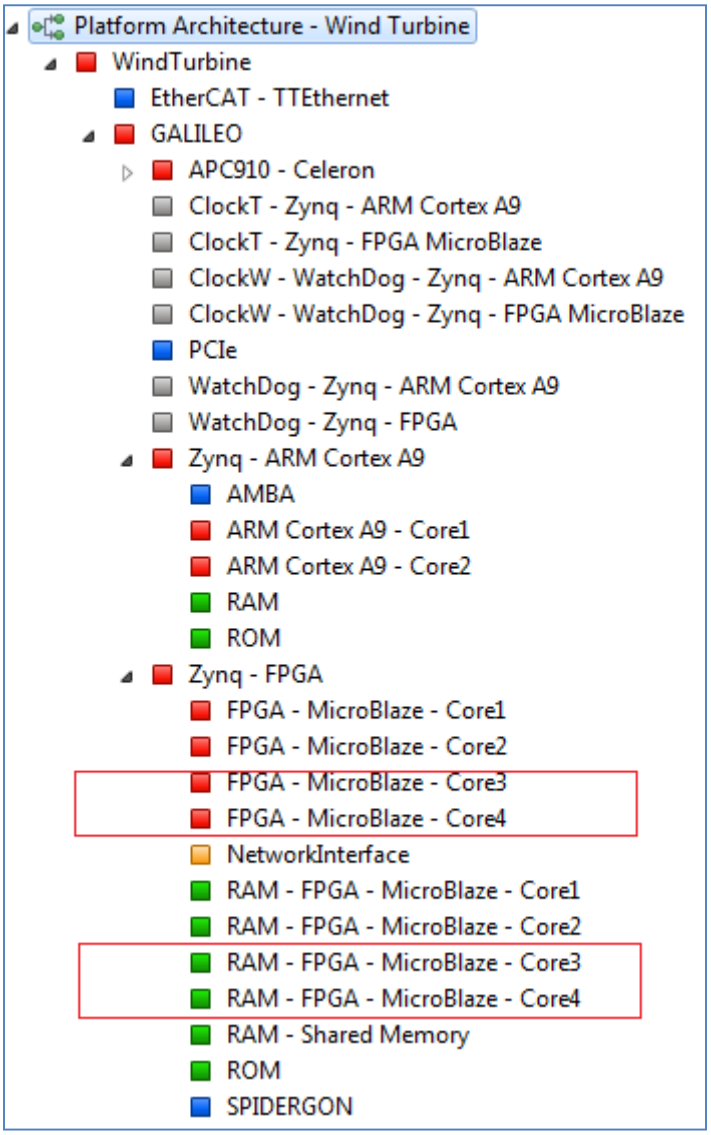

Figure 21. Additional cores and memory when extra supervision is required.

In the variability resolution phase, the user chooses the "Supervision Plus" feature. This yields two additional valid products described in the following sections.

4.2.3.1 Wind Power without Supervision Plus

In this case, no extra core neither partition is created. Figure 22 below shows the impact on the component architecture. The safety model is not affected as the supervision has no safety requirements.

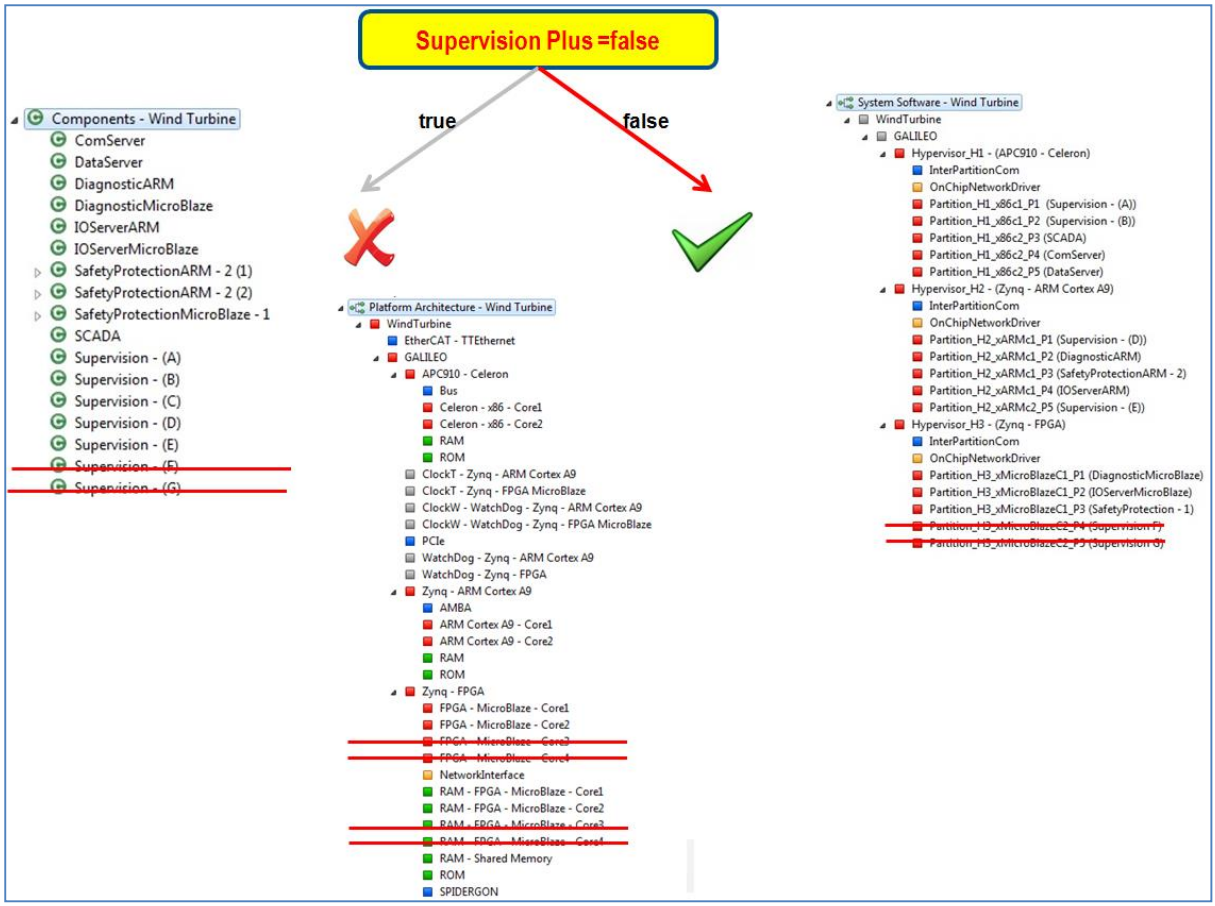

Figure 22. Supervision function with no extra components – impact in models

4.2.3.2 Wind Power with Supervision Plus

In this case, two extra cores and two extra partitions are created. Figure 23 below shows the impact on the component architecture. The safety model is not affected as the supervision has no safety requirements.

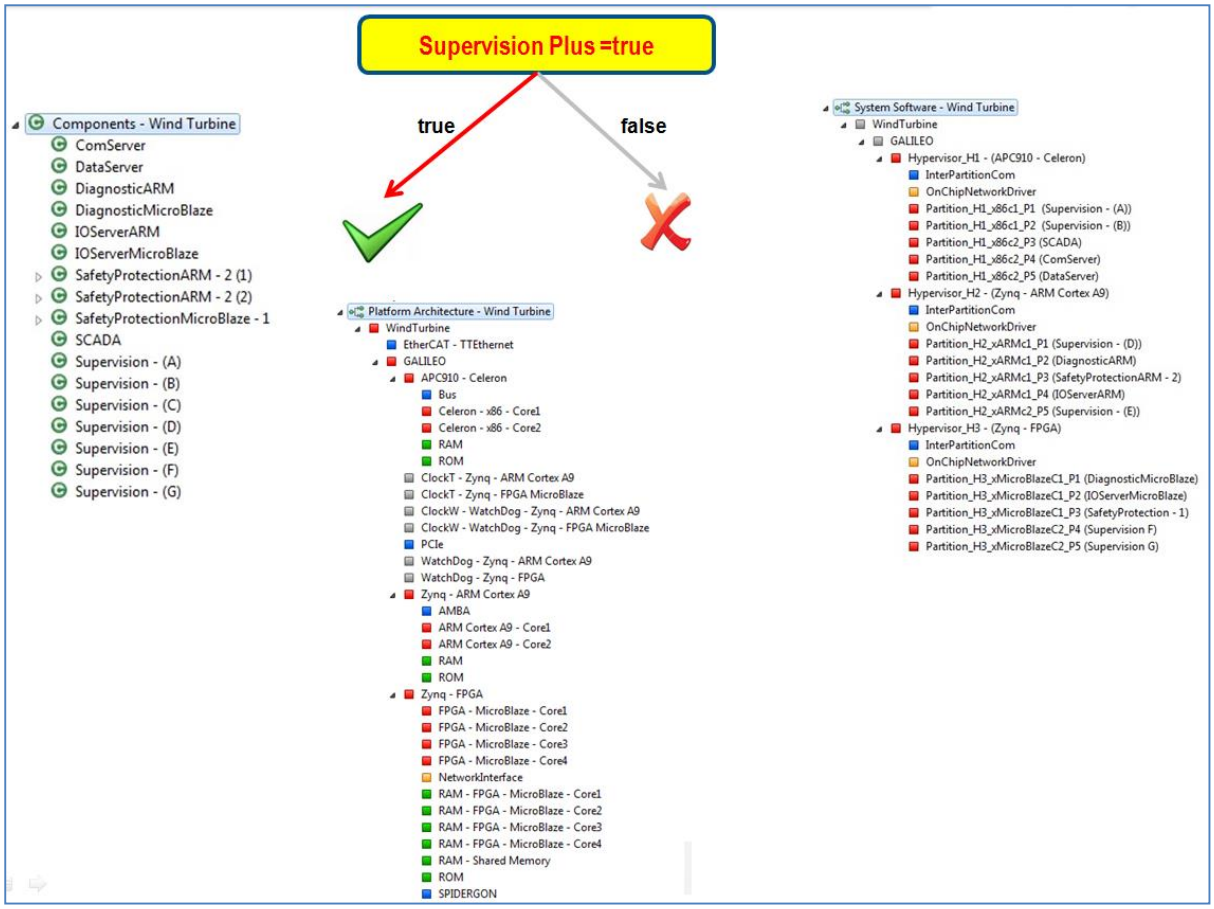

Figure 23. Supervision function with extra Components – impact in models

However, note that the extra components are not allocated in the two extra cores but in the second core.

4.2.4 Combinatorial Explosion and Need of Combinatorial Testing

As for the wind power test case, up to 9 features have been identified so far:

- Safety Protection: *redundant* and *diverse*, giving rise to 3 possible configurations
- Speed Sensor: *redundant* and *diverse*, giving rise to 3 possible configurations
- Vibration Sensor: *redundant* and *diverse*, giving rise to 3 possible configurations
- Voltage Sensor: *redundant* and *diverse*, giving rise to 3 possible configurations
- Supervision Plus: giving rise to 2 possible configurations

Although some of them are independent form each other, strictly speaking there is a total number of 3 x (3 x 3 x 3) x 2 configurations, so *162* possible configurations. Obviously, it is not possible to verify these 162 configurations, so specific techniques are needed to cope with combinatorial explosion.

4.3 Deployment Feasibility

A deployment is feasible if a valid schedule of components exists, if communication routes exist, if all safety constraints are met, if all timing constraints are met, etc. Many properties must be verified to accept a given deployment as valid.

In this section, as an example, a subset of the (i) safety constraints and (ii) rule checking for wind power use case is given so that the reader can understand how a deployment is verified from the safety point of view.

4.3.1 Safety Allocation Constraints

Once the variability is resolved (i.e., the features are chosen), the specific models are generated. Taking the specific models as input, a number of sets of safety constraints for deployments are generated. Figure 24 below shows some of the main sets of constraints for components and partitions (corresponding to some verification documented in Section 3.5.1.1).

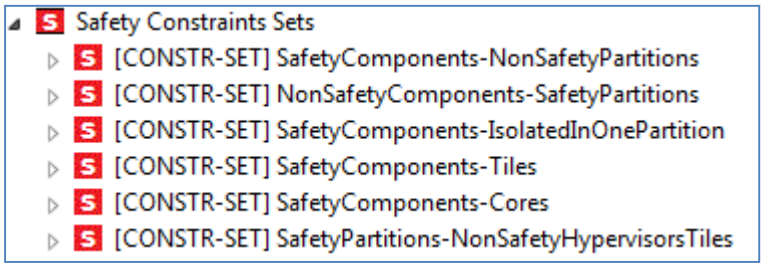

Figure 24. Example of Safety Constraints Sets

These constraints, along with the rule checking, will be used to verify if a deployment is valid from the safety point of view (i.e., whether it violates safety requirements with the information available).

These constraints are evaluated during the design space exploration process, which validates the produced deployments using the safety constraint rules described in this section.

4.3.1.1 Safety Components Not In Non Safety Partitions

This group of constraints verifies that the deployment has not allocated a safety component to a non-safety partition. If any of these constraints is violated, the deployment is discarded.

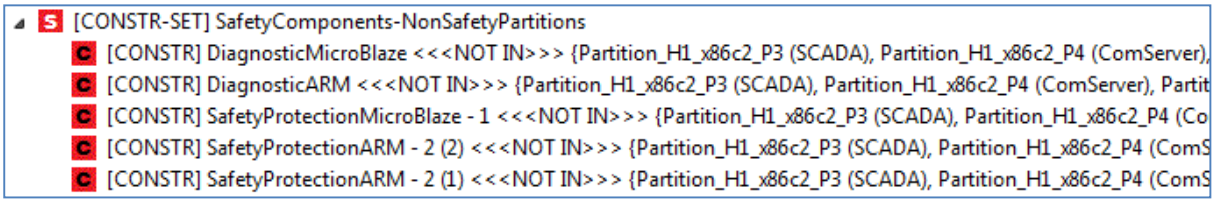

Figure 25. Safety Constraints Set example

For example, the Safety ProtectionMicroBlaze-1 application cannot be deployed into the non-safety partitions listed after the “NOT IN” clause.

4.3.1.2 Non Safety Components Not In Safety Partitions

This group of constraints verifies that the deployment has not allocated a non-safety component to a safety partition. If any of these constraints is violated, the deployment is discarded.

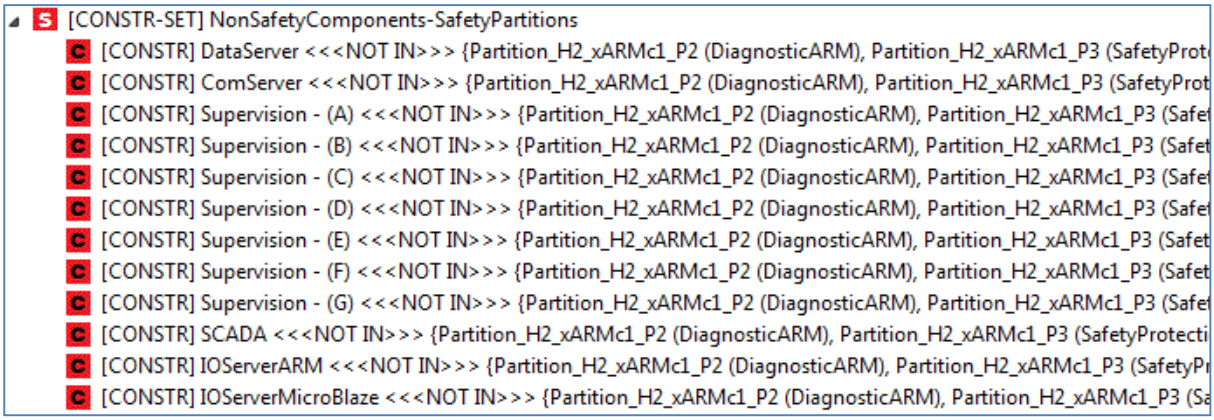

Figure 26. Safety Constraint Set example

For example, Supervision-(A) application cannot be deployed into safety partitions, listed in the “NOT IN” list of partitions.

4.3.1.3 Safety Components Isolated In One Partition

This group of constraints verifies that, when a safety component requires being alone in its own partition, the deployment has not allocated any other component in its partition.

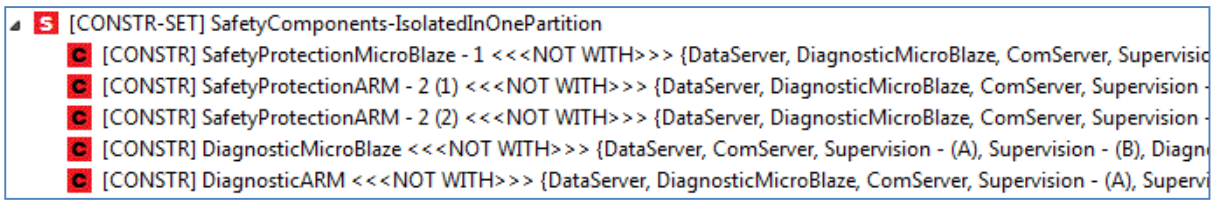

Figure 27. Safety Constraint Set example

In this case, for each safety component, a constraint saying that it can NOT be deployed WITH any other application (safety or no safety) is generated.

4.3.1.4 Safety Components in a Given Tile

This group of constraints verifies that when a safety component requires being allocated in a partition allocated in a hypervisor of a given tile (regardless of the core), the deployment has not allocated its partition to any other tile.

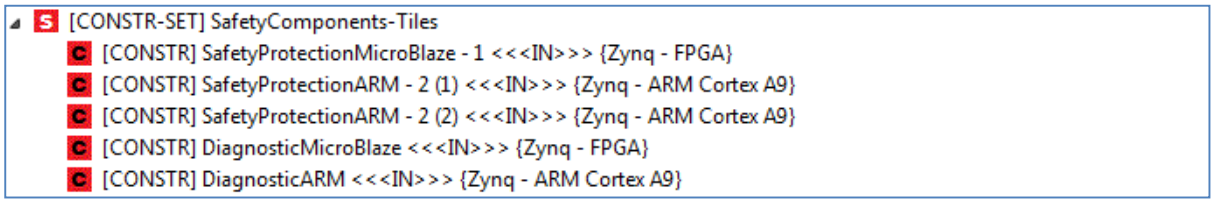

Figure 28. Safety Constraint Set example

For example, DiagnosticARM application is forced to be allocated in ARM Cortex A9 processor.

4.3.1.5 Safety Components in a given Core

This group of constraints verifies that when a safety component requires being allocated in a partition allocated in a given core, the deployment has not mapped its partition to any other core.

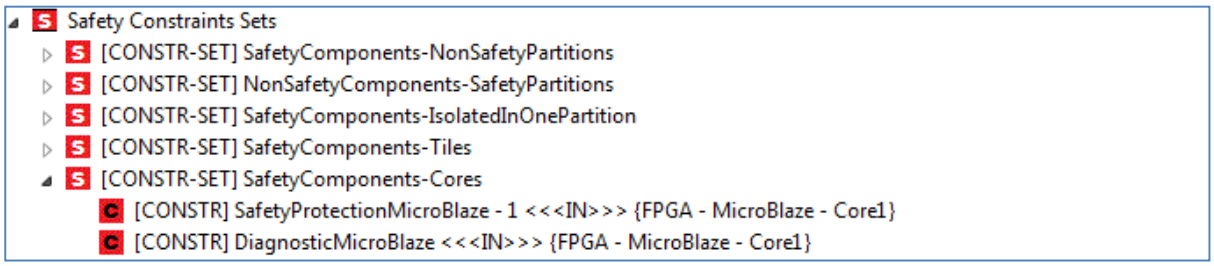

Figure 29. Safety Constraint Set example

For example, the DiagnosticMicroBlaze application is forced to be allocated in Core1 of FPGA MicroBlaze1 processor.

4.3.2 Safety Consistency Rules Checking

As an example of safety rule checking, the wind power use case hardware platform specifies that each tile will have a diagnosis technique with a watchdog (Technique A.9.2 of Table A.10 of IEC 61508-2 [17]) as shown in Figure 30 below.

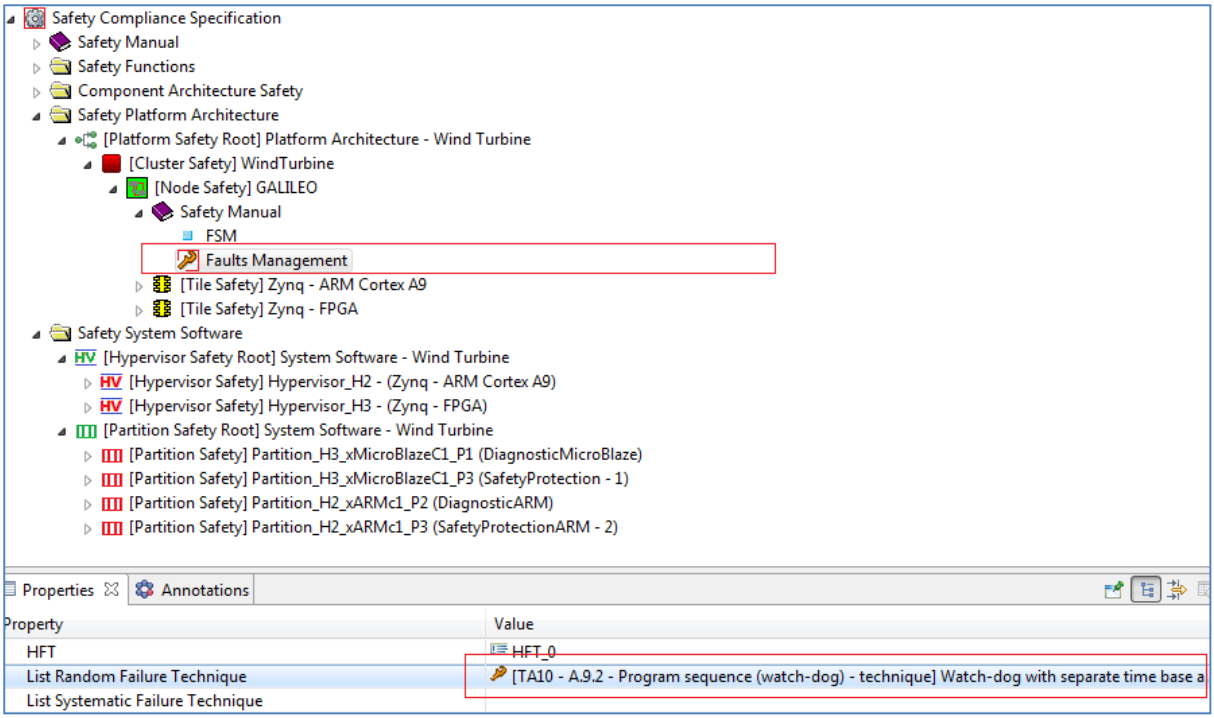

Figure 30 GALILEO Node Safety Manual specifying a Watchdog-based technique in its Safety Manual

The corresponding safety rule will check (as described in section 3.5.1.2) among other things that the hardware platform, in addition to having three watchdogs has six independent clocks, three for the tiles and three for the watchdogs as shown in Figure 31 below.

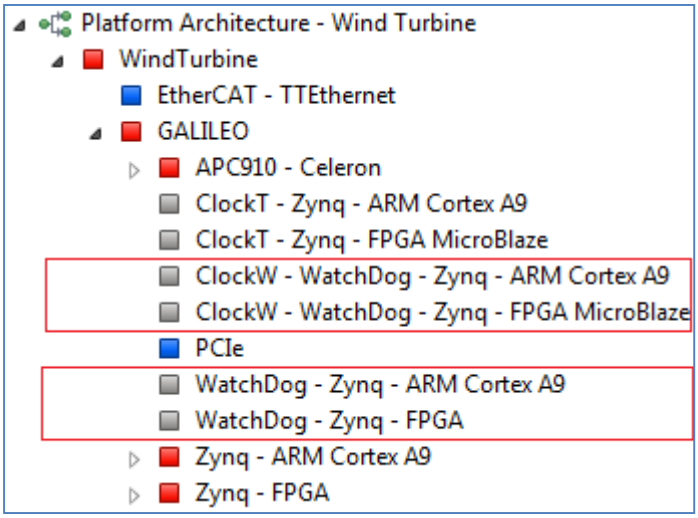

Figure 31 WP7 Wind Power platform Watchdogs and Clocks

In addition, these clocks are connected to the corresponding tiles and watchdogs. The rule also checks that the diagnostic components have access rights to the corresponding watchdog (as shown in Figure 32 below) so that partitions is given access rights.

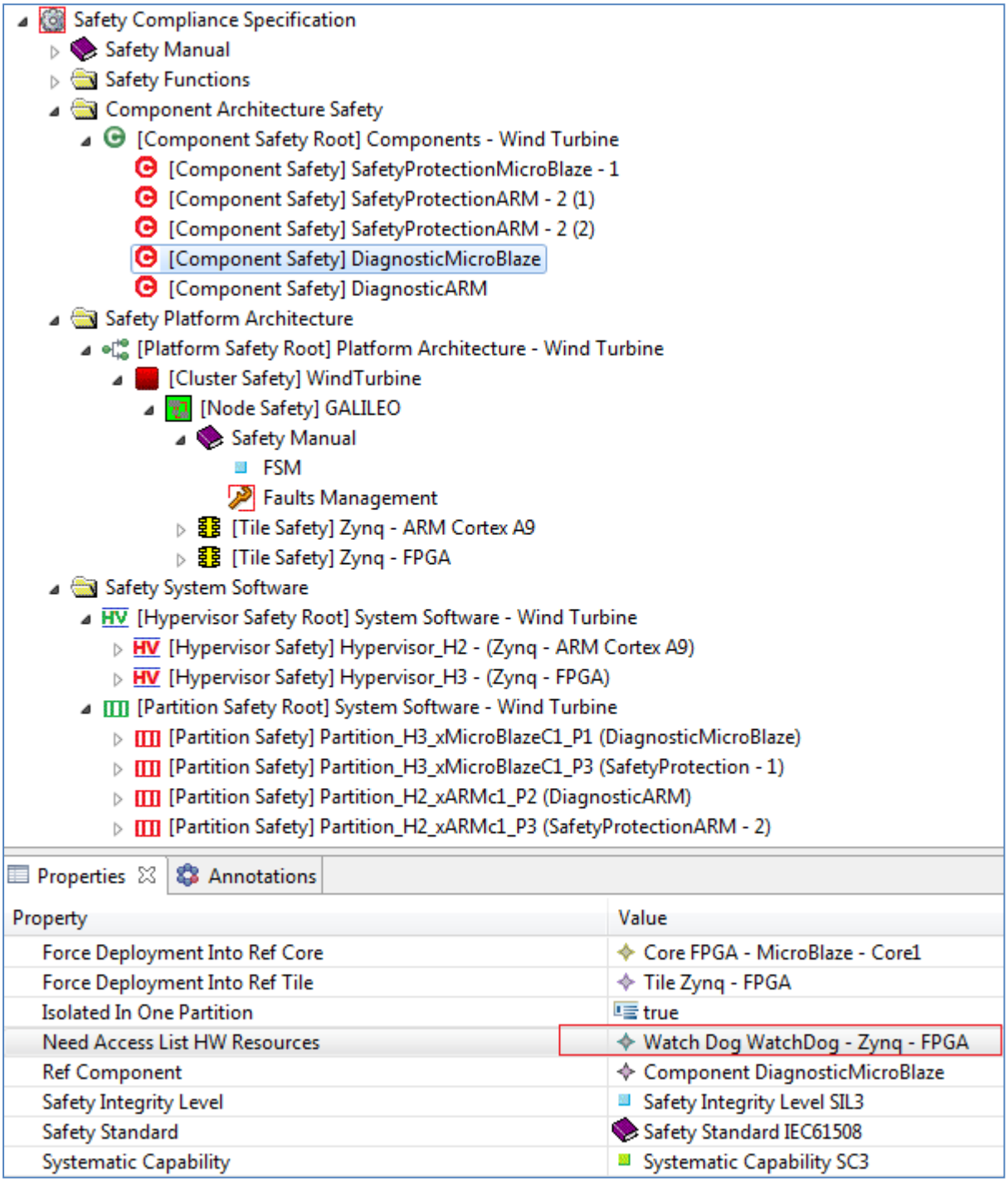

Figure 32. Diagnostic APC910-Celeron component must have access rights to its Watchdog

4.3.3 Evidences for Certification in WP5

Finally, we shall remember that many of the verifications will produce trace documents in the appropriate format that will be used in the certification methodology to be defined in Task 5.5 D 5.5.3. This methodology aims at helping in the process of certification of mixed-criticality multi-core systems.

As part of that methodology, deliverables D 5.1.1 (see pages 45, 46) submitted and positively assessed by TÜV (safety cases for COTs processors, Hypervisor, Partition) define the *claims-arguments-evidences* for certification of COTS processors, Hypervisor and Partitions. In the same way, *a safety certification strategy for IEC 61508 compliant industrial mixed-criticality systems based on multicore partitioning [48]* was developed as part of FP7 MultiPARTES project.

All those safety cases define *claims-arguments-evidences* for certification. The documents generated by WP4 verification tools will be part of these “evidences”. The exact format of the required documents (to be presented as evidences) will be defined in that task.

5 Conclusion

The purpose of this deliverable was to "provide an overview and assessment of different analysis and testing techniques that may be useful for mixed criticality systems with explicit variability models".
We reviewed the challenges and verification under variability, and especially the issue of feature interaction and how it can be approached using combinatorial interaction testing. We also reviewed the DREAMS meta-models, showing where variability can be specified and how it can impact extra-functional concerns such as reliability, power consumption or safety. We then described our approach to Task 4.3, which "*defines means to resolve the explicit variability's*". Our solution combines techniques from product lines with evolutionary optimization to address both business and technical variability. Business variability is first documented using BVR feature models, while product sampling is used to automatically resolve the variability related to the features offered by the system, and to produce the associated AutoFOCUS3 specification. These models are then feed into the design space exploration tool, where technical variability regarding implementation is resolved using evolutionary optimization.
The two following deliverables due in Task 4.3 will provide a prototype implementation of this tool chain that will be later evaluated on the wind power demonstrator in WP7.

6 Bibliography

[1] DREAMS Consortium, "Distributed REal-Time Architecture for Mixed Criticality Systems (DREAMS) - Description of Work," ed: EU Seventh Framework Programme, 2013.

[2] S. Barner, A. Diewald, F. Eizaguirre, Ó. Saiz, L. Havet, J. Migge*, et al.*, "Meta-models for Application and Platform," DREAMS Consortium D1.4.1, 2014.

[3] Ø. Haugen, F. Chauvel, A. Larrucea, J. Perez, S. Trujillo, and V. Brocal, "State of the Art of Piecewise Certification of Mixed Criticality Systems," DREAMS Consortium D5.5.1, 2014.

[4] A. Syed, G. Fohler, A. Agrawal, F. Chauvel, A. M. Diewald, S. Barner*, et al.*, "Definition of Offline Adaptation Strategies for Mixed-Criticality and Initial Implementation," DREAMS Consortium D4.1.2, 2015.

[5] S. Apel and C. Kästner, "An Overview of Feature-Oriented Software Development," *Journal of Object Technology* vol. 8, pp. 49-84, 2009.

[6] Ø. Haugen, "BVR - The Language," VARIES Consortium D4.2, 2014.

[7] Object Management Group, "Unified Modeling Language (UML) v2.4.1 - Superstructure Specification," ed: Object Management Group, 2011.

[8] Object Management Group, "Unified Modeling Language (UML) v2.4.1 - Infrastructure Specification," ed: Object Management Group, 2011.

[9] Object Management Group, "OMG Systems Modeling Language (OMG SysML)," ed: Object Management Group, 2012.

[10] F. Hölzl and M. Feilkas, "AutoFocus 3: a scientific tool prototype for model-based development of component-based, reactive, distributed systems," presented at the Proceedings of the 2007 International Dagstuhl conference on Model-based engineering of embedded real-time systems, Dagstuhl Castle, Germany, 2010.

[11] H. Thompson, B. Bauer, B. Boeddeker, I. Broster, M. Coppola, A. Crespo*, et al.*, "Report from the Workshop on Mixed Criticality Systems," European CommisionFebruary 2012.

[12] IEC, "Functional safety of electrical/electronic/programmable electronic safety-related systems," in *Part 7: Overview of techniques and measures* vol. 61508-7:2010, ed, 2010.

[13] IEC, "Functional safety of electrical/electronic/programmable electronic safety-related systems," in *Part 6: Guidelines on the application of IEC 61508-2 and IEC 61508-3* vol. 61508-6:2010, ed, 2010.

[14] IEC, "Functional safety of electrical/electronic/programmable electronic safety-related systems," in *Part 5: Examples of methods for the determination of safety integrity levels* vol. 61508-5:2010, ed, 2010.

[15] IEC, "Functional safety of electrical/electronic/programmable electronic safety-related systems," in *Part 4: Definitions and abbreviations* vol. 61508-4:2010, ed, 2010.

[16] IEC, "Functional safety of electrical/electronic/programmable electronic safety-related systems," in *Part 3: Software requirements* vol. 61508-3:2010, ed, 2010.

[17] IEC, "Functional safety of electrical/electronic/programmable electronic safety-related systems," in *Part 2: Requirements for electrical/electronic/programmable electronic safety-related systems* vol. 61508-2:2010, ed, 2010.

[18] IEC, "Functional safety of electrical/electronic/programmable electronic safety-related systems," in *Part 1: General requirements* vol. 61508-1:2010, ed, 2010.

[19] IEC, "Functional safety of electrical/electronic/programmable electronic safety-related systems (Second edition)," vol. 61508:2010, ed, 2010.

[20] RTCA/EUROCAE, "Software Considerations in Airborne Systems and Equipment Certification," vol. DO-178B/ED-12B, ed, 1992.

[21] RTCA, "Design Assurance Guidance for Airborne Electronic Hardware," vol. DO-254, ed, 2000.

[22] ISO, "Road vehicles — Functional safety," in *Part 9: Automotive Safety Integrity Level (ASIL)-oriented and safety-oriented analyses* vol. 26262-9:2011, ed, 2011.

[23] ISO, "Road vehicles — Functional safety," in *Part 7: Production and operation* vol. 26262-7:2011, ed, 2011, pp. vi,10.

[24] ISO, "Road vehicles — Functional safety," in *Part 6: Product development at the software level* vol. 26262-6:2011, ed, 2011, pp. viii,46.

[25] ISO, "Road vehicles — Functional safety," in *Part 5: Product development at the hardware level* vol. 26262-5:2011, ed, 2011, pp. viii,61.

[26] ISO, "Road vehicles — Functional safety," in *Part 4: Product development at the system level* vol. 26262-4:2011, ed, 2011, pp. viii,37.

[27] ISO, "Road vehicles — Functional safety," in *Part 3: Concept phase* vol. 26262-3:2011, ed, 2011, pp. vi,25.

[28] ISO, "Road vehicles — Functional safety," in *Part 2: Management of functional safety* vol. 26262-2:2011, ed, 2011, pp. vi,26.

[29] ISO, "Road vehicles — Functional safety," in *Part 8: Supporting processes* vol. 26262-8:2011, ed, 2011, pp. viii,46.

[30] ISO, "Road vehicles — Functional safety," in *Part 1: Vocabulary* vol. 26262-1:2011, ed, 2011, pp. vi,20.

[31] ISO, "Road vehicles — Functional safety," vol. 26262:2011, ed, 2011.

[32] J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, and G. Saake, "An overview on analysis tools for software product lines," presented at the Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2, Florence, Italy, 2014.

[33] J. M. Jézéquel and B. Meyer, "Design by contract: the lessons of Ariane," *Computer,* vol. 30, pp. 129-130, 1997.

[34] J. H. Miller and S. E. Page, *Complex Adaptive Systems: An Introduction to Computational Models of Social Life*: Princeton University Press, 2007.

[35] B. J. Garvin and M. B. Cohen, "Feature Interaction Faults Revisited: An Exploratory Study," in *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*, 2011, pp. 90-99.

[36] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, Jr., "Software fault interactions and implications for software testing," *Software Engineering, IEEE Transactions on,* vol. 30, pp. 418-421, 2004.

[37] M. Steffens, S. Oster, M. Lochau, and T. Fogdal, "Industrial evaluation of pairwise SPL testing with MoSo-PoLiTe," presented at the Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems, Leipzig, Germany, 2012.

[38] M. B. Cohen, M. B. Dwyer, and S. Jiangfan, "Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach," *Software Engineering, IEEE Transactions on,* vol. 34, pp. 633-650, 2008.

[39] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surv.,* vol. 43, pp. 1-29, 2011.

[40] M. F. Johansen, "Testing Product Lines of Industrial Size: Advancements in Combinatorial Interaction Testing," PhD in Computer Science, Faculty of Mathematics and Natural Sciences, University of Oslo, Oslo, 2013.

[41] R. R. Mukkamala, A. Wąsowski, T. Berger, A. F. Iosif-Lazăr, Ș. Stănciulescu, R. Haemmerle*, et al.*, "Variability Analysis Solutions," VARIES Consortium D4.7, 2015.

[42] Object Management Group, "Meta Object Facility (MOF) Core - v2.4.1," ed: Object Management Group, 2014.

[43] RTCA/EUROCAE, "Software Considerations in Airborne Systems and Equipment Certification," vol. DO-178C/ED-12C, ed, 2011.

[44] M. Črepinšek, S.-H. Liu, and M. Mernik, "Exploration and exploitation in evolutionary algorithms: A survey," *ACM Comput. Surv.,* vol. 45, pp. 1-33, 2013.

[45] J. Perez, A. Larrucea, T. Trapman, V. Brocal, J. Coronel, F. Chauvel*, et al.*, "Modular Safety Case for Hypervisor," DREAMS Consortium D 5.1.1, January 2015.

[46] J. Perez, A. Larrucea, B. Nafria, I. Canales, T. Trapman, V. Brocal*, et al.*, "Modular Safety Case for COTS processor " DREAMS Consortium D 5.1.2, 2015.

[47] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *Dependable and Secure Computing, IEEE Transactions on,* vol. 1, pp. 11-33, 2004.

[48] J. Perez, D. Gonzalez, C. F. Nicolas, T. Trapman, and J. M. Garate, "A Safety Certification Strategy for IEC-61508 Compliant Industrial Mixed-Criticality Systems Based on Multicore Partitioning," in *Digital System Design (DSD), 2014 17th Euromicro Conference on*, 2014, pp. 394-400.

Appendix A Safety Verification Constraints

This appendix shows some examples of safety verification constraints that are checked for every deployment that DSE explores. In Deliverable 4.3.2, the final list of constraints to be implemented will be defined and detailed.

A.1 Safety Constraints for Components

A.1.1 Constraint - Safety Components only into Safety Partitions

Semantics

Safety components can only be allocated in safety partitions. In addition, the theoretical SIL of a partition must be greater or equal than the SIL claimed by each of its components.

Related Variability in Meta-models

Depending on the value of a given feature (or combination of values of a set of features), after resolving variability, some components or partitions can be eliminated. Resulting safety partitions must be able to accommodate resulting safety components.

Impact of variability in Safety Function

Some combinations of features can lead to impossible configurations, in which there is no way to allocate safety components to safety partitions. The tool must ensure that these combinations of features are ruled out.

A.1.2 Constraint - Non Safety Components only into Non Safety Partitions

Semantics

Ensure that non-safety components are only allocated to non-safety partitions, whose SIL level is sufficient.

Related Variability in Meta-models

Depending on the value of a given feature (or combination of values of a set of features), after resolving variability, some components or partitions can be eliminated. The resulting partitions must be able to accommodate resulting components.

Impact of Variability in Safety Function

Violations of this constraint may lead to impossible configurations where a non-safety component allocated to a safety partition would prevent guaranteeing the SIL levels for safety components. The tool should ensure that these combinations of features are ruled out.

A.1.3 Constraint - Safety Components Isolated in One Partition

Semantics

The system architect may want that a safety component be isolated in one safety partition (i.e., only one safety component is allocated per safety partition). This is the case of all the safety components in the wind power use case, as shown in the Figure 33 below.

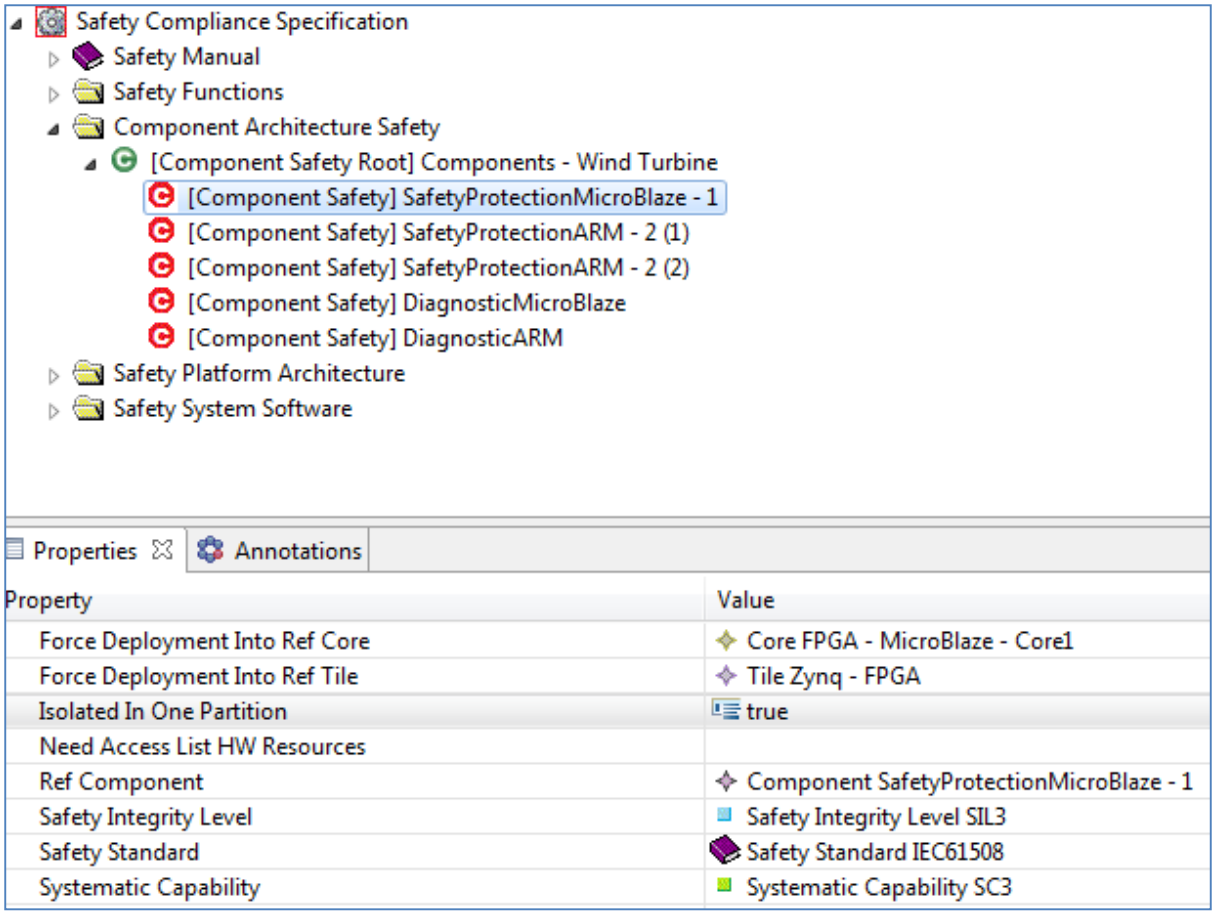

Figure 33. Safety Component requirement: Isolation in one partition

For each allocated component, it is checked that, if the component is a safety component that must be isolated (see Property *IsolatedInPartition, set to True)* then, no other component in the deployment is allocated in that partition.

Related Variability in Meta-models

Theoretically, depending on the value of a given feature (or combination of values of a set of features), the safety model may:

- Specify this property to true or false for a given component.
- Resulting safety partitions may be different

The resulting safety partitions must be able to accommodate the safety components satisfying the required properties.

Impact of Variability in Safety Function

Some combinations of features can lead to impossible configurations, where there is no way to allocate safety components to safety partitions. The tool shall ensure that these combinations of features are ruled out.

A.1.4 Constraint - Safety Components Specifying Tiles/Cores

Semantics

The system architect may want to specify that a safety component be allocated in a partition allocated in a given tile (regardless of the core). The wind power use case includes such safety components, as shown in Figure 34 below.

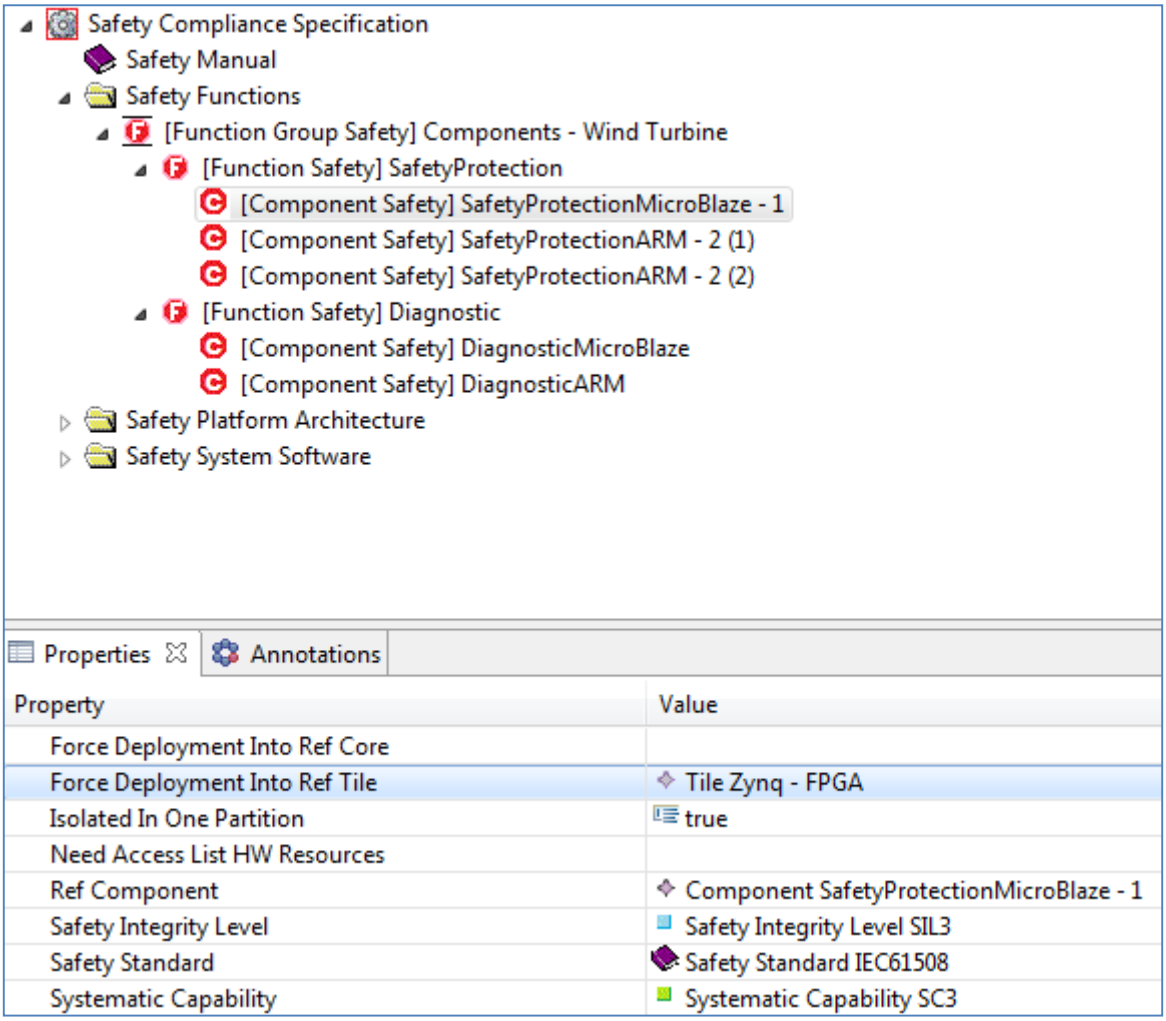

Figure 34. Safety Component requirement: Execution in a given partition-hypervisor-tile

Verification Procedure

For each allocated component, it is checked that if the component is a safety component and specifies a given tile/core then the component is allocated in a partition that is allocated in a core belonging to that tile, or, if a core is specified, in a partition allocated in that core.

Related variability in Meta-models

Theoretically, depending on the value of a given feature (or combination of values of a set of features), the safety model may:

- Specify tiles/cores for allocation of components.
- Resulting safety partitions may be different

The resulting safety partitions must be able to accommodate the resulting safety components satisfying the required properties.

Impact of Variability in Safety Function

Some combinations of features can lead to impossible configurations, in which there is no way to allocate safety components to safety partitions. The tool shall ensure that these combinations of features are ruled out.

A.1.5 Constraint - SIL claimed by Component supported by Partition

Semantics

If a safety component claims a given SIL level, then the deployment must allocated it into a safety partition with a greater or equal SIL.

Related variability in Meta-models

SIL claims may vary in function of features.

Impact of Variability in Safety Function

Some combinations of features can lead to impossible configurations, where there is no way to allocate safety components to safety partitions. The tool shall ensure that these combinations of features are ruled out.

A.1.6 Constraint - All Components in a Partition have the same SIL

Semantics

If a safety partition hosts more than one safety component, then the SIL level of each component is equal to the lowest SIL level of all components. As a consequence, all will have the same SIL level.

Related Variability in Meta-models

SIL claims may vary in function of features, as the resulting safety partitions.

Impact of variability in Safety Function

Some combinations of features can lead to impossible configurations, where there is no way to allocate safety components to safety partitions, while respecting all SIL requirements. The tool shall ensure that these combinations of features are ruled out.

A.2 Safety Constraints for Partitions

In the DREAMS platform meta-model, the number of partitions and the allocation of partitions to hypervisors are fixed in the model, although it may vary with features. Then, although they are not strictly part of the deployment, the safety model acts as if partitions were mapped to hypervisors by the deployment and therefore, verifies its validity from the safety point of view.

As a consequence, a set of constraints is generated for this purpose, and described in the following subsections.

A.2.1 Constraint - Safety Partitions only into Safety Hypervisors

Semantics

Safety partitions can only be allocated in safety hypervisors.

Related Variability in Meta-models

In theory a partition is independent from the hypervisor and can be allocated in different hypervisors.

Impact of Variability in Safety Function

Partition SIL claims heavily depend on the possibility of being allocated on a safety hypervisor supporting a higher SIL. The tool shall ensure that erroneous combinations of features leading to impossible mappings are ruled out.

A.2.2 Constraint - Safety Partitions only into Safety Cores

Semantics

Safety partitions must be allocated into cores that belong to safety tiles (belonging to nodes that specify a safety manual in the safety model).

Related Variability in Meta-models

In theory, a partition is independent from its underlying hypervisor, and the hypervisor can be allocated in different tiles.

Impact of Variability in Safety Function

A partition SIL claim heavily depends on the possibility of being allocated on a safety hypervisor supporting a higher SIL. The tool shall ensure that erroneous combinations of features leading to impossible mappings are ruled out.

A.2.3 Constraint - SIL claimed by Partition supported by Hypervisor

Semantics

If a safety partition claims a given SIL level, then only a safety hypervisor, whose SIL is greater or equal, can host this component. Note that this is an error depending on the SIL level required by the components allocated in the partition.

Related Variability in Meta-models

SIL claims may vary in function of features.

Impact of Variability in Safety Function

Some combinations of feature can lead to impossible configurations, where there is no way to allocate safety partitions to safety hypervisors while respecting all SIL claims. The tool should ensure that these combinations of features are ruled out.

A.3 Safety Constraints for Hypervisors

A.3.1 Constraint - Safety Hypervisor only into Safety Tiles

Semantics

Safety hypervisors must be allocated in safety tiles.

Related Variability in Meta-models

As mentioned before, in theory, a hypervisor can be allocated in different tiles.

Impact of variability in Safety Function

Supporting a hypervisor SIL claim heavily depends on the possibility of being allocated on a safety tile ensuring a higher or equal SIL. The tool shall ensure that erroneous combinations of features leading to impossible mappings are rule out.

A.3.2 Constraint - SIL claimed by Hypervisor Supported by Tile

Semantics

If a safety hypervisor claims a given SIL level, then the deployment must allocate it into a safety tile claiming a greater or equal SIL. Note that this is an error depending on the SIL level required by the components allocated to the partitions allocated to the hypervisor.

Related Variability in Meta-models

SIL claims may vary in function of features.

Impact of Variability in Safety Function

Again, some combinations of features can lead to impossible configurations where there is no way to allocate the safety hypervisor to safety tiles while respecting SIL claims. The tool shall ensure that these combinations of features are ruled out.

Appendix B Safety Verification Rules

This appendix shows some examples of safety verification *rules* that are checked when the whole product is generated. In Deliverable 4.3.2, the final list of rules to be implemented will be detailed.

For the purpose of clarity, we have divided the examples into two groups: *generic verification rules* and more *expert verification rules*.

B.1 Generic verification rules

B.1.1 Rule - SIL claimed cannot be higher than the maximum allowable SIL

SIL level claimed to a safety compliant item (SCI) such as a component, a node, a tile, a hypervisor or partition, cannot be higher than the allowable SIL value calculated depending on the diagnostics used for that compliant item.

If a given SCI claims a certain SIL level, it has to be checked that the allowable SIL value calculated depending on the diagnostic techniques used and declared in the safety manual justifies a level that is equal or higher than the level claimed for the item. Otherwise, it cannot be supported the SIL claim of the SCI.

The following illustrates this rule. Suppose that the hardware platform has a watchdog to reset the processor when necessary. The safety model (see Figure 35) specifies the following HFT level and the following diagnostic technique:

- HFT = 0
- Diagnostic Technique A.9.2 “Watch-dog with separate time base and time-window” that has a medium diagnostic coverage (DC).

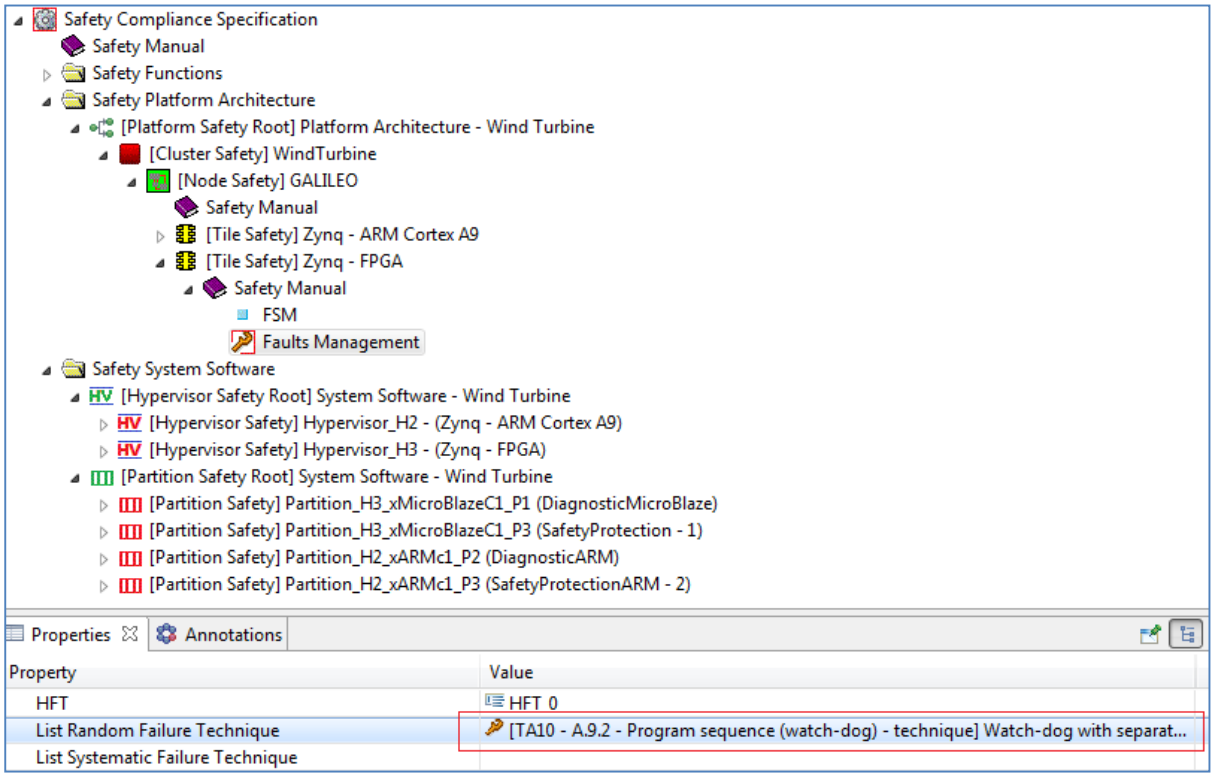

Figure 35 Watchdog based Diagnosis Technique Specification

According to Table 3 in IEC 61508 [17] a HTF of 0 coupled to a medium DC has a maximum allowable SIL of 1. However, for the hardware platform, a SIL of 3 is claimed (not displayed in Figure 35) and

therefore an error is given as this SIL level cannot be guaranteed with this rule for the watchdog. This rule is checked for every safety manual and every diagnostic technique defined in the manual.

B.1.2 Rule - Safety certification standard supported by any 'compliant item' must be compliant with the system certification standard

All the components must be compliant with the certification standard of the system. If any component is not compliant, the system will not be considered consistent. If any component is not compliant with system's standard, but supports a derived standard, it will be considered consistent, but a warning will be issued.

When in a given system, there is at least one component that is not compliant with the certification standard, then, the whole system cannot be certified by that standard. In the case that the non-conformance component is certified by a derived standard (for example ISO 26262 [22-31] derived from IEC 61508 [12-19]), then the whole system is consistent but a warning is given indicating that a derived standard is used for a given component must be raised.

B.1.3 Rule - FSM used in the development of any compliant item must be compliant with the system FSM defining a SIL level grater or equal than FSM SIL level for the whole system

The functional safety management (FSM) – a collection of the supported safety integrity levels by SCI, safety compliant item – must be compliant with the FSM for the whole system. The FSM used in the development of any SCI must be compliant with the system FSM defining a SIL level greater or equal than the FSM SIL level for the whole system. The FSM for each component must be compliant with the FSM for the system.

The rule must ensure that when the HFT is 0, the system's SIL level is not greater than any of the SIL levels of composing SCI node's SIL level.

B.2 Expert Verification Rules

This section describes a type of rules that we call *expert validation rules.* They capture more complex aspects of IEC 61508 [12-19], which impose tight constraints on hardware architectural layouts.

B.2.1 Rule – HW Architecture Required By a Watchdog Table A10 - Technique Diagnosis A.9.1 and A.9.2

If a watchdog is defined in a platform and diagnosis techniques (see A.9.1 or A.9.2 of Table A.10 of IEC 61508 [12-19]) are used to guarantee the SIL level, then, the following architectural requirements must be met by the platform:

- The platform has a watchdog resource defined and connected to the corresponding processor(s) of the platform.
- The platform has two different clocks resources defined.
- One clock is connected to the watchdog and the other clock is connected to the processor(s)
- There is at least one component that has access to the watchdog resource

B.2.2 Rule – Variable Memory Ranges TableA6 - Technique Diagnosis A.5.1 to A.5.7

For each safety partition, one of the techniques of Table A.6 of Variable Memory [12-19], ranges have to be selected. Implementation is straightforward.

B.2.3 Rule – Variable Memory Ranges Double RAM Table A6 - Technique Diagnosis A.5.7

For each safety partition, if the variable memory ranges technique selected is A.5.7, then the platform hardware architecture must have two RAM resources defined.

The A.5.7 diagnosis technique of IEC 61508 [12-19] implies that the platform has two RAM blocks (of equal size) defined. This is exactly what is checked by this rule, producing an error if only one RAM block is found.