Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Bacharelado em Ciência da Computação

A Reuse Oriented Approach to Support the Engineering of Safety-Critical Systems using the CHESS Toolset

Lucas Paiva Bressan

JUIZ DE FORA

DEZEMBRO, 2018

A Reuse Oriented Approach to Support the Engineering of Safety-Critical Systems using the CHESS Toolset

Lucas Paiva Bressan

Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Bacharelado em Ciência da Computação

Orientador: André Luiz de Oliveira

Coorientador: Barbara Gallina

JUIZ DE FORA

DEZEMBRO, 2018

A Reuse Oriented Approach to Support the Engineering of Safety-Critical Systems using the CHESS Toolset

Lucas Paiva Bressan

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM CIÊNCIA DA COMPUTAÇÃO.

Aprovada por:

JUIZ DE FORA

3 DE DEZEMBRO, 2018

Resumo

Sistemas críticos são sistemas de software nos quais falhas podem levar a consequências catastróficas, que variam desde danos ambientais, financeiros, à propriedade até lesões e perda de vidas humanas. Em virtude de sua natureza crítica, as propriedades de segurança desses sistemas devem ser analisadas e verificadas em diferentes níveis de abstração. Assim, atividades de engenharia de segurança como Hazard Analysis and Risk Assessment (HARA), Fault Tree Analysis (FTA) e Failure Modes and Effects Analysis (FMEA) devem ser realizadas para identificar as potenciais ameaças à segurança do sistema e a propagação de potenciais falhas pelos seus componentes. A realização dessas atividades é necessária para a produzir artefatos requeridos por padrões de segurança e autoridades para certificação e liberação do sistema para operação. Entretanto, a produção de artefatos de engenharia de segurança impacta no aumento significativo dos custos e esforços de projeto. Técnicas orientadas a reúso como Engenharia de Linha de Produtos de Software, juntamente com técnicas dirigidas a modelos, vêm sendo amplamente adotadas pela indústria no desenvolvimento de sistemas críticos por proporcionar o aumento da produtividade, reduzir os custos de produção de artefatos de projeto e de engenharia de segurança e o tempo de entrega do produto final. Existem no mercado, um conjunto de ferramentas de apoio à engenharia de sistemas críticos, dentre elas, MATLAB Simulink e HIP-HOPs, CHESS e OSATE AADL & Error Annex. Essas ferramentas visam apoiar o desenvolvimento de modelos arquiteturais e de erros de sistemas críticos. Ferramentas como CHESS fornecem apoio ao projeto, atividades de engenharia de segurança e geração de código para sistemas críticos. Apesar do nível de maturidade de CHESS, existe atualimente, uma falta de diretrizes para promover o seu uso adequado no desenvolvimento de sistemas críticos. Além disso, a falta de integração do CHESS com ferramentas de apoio ao reúso, como o Base Variability Resolution (BVR), impõe um obstáculo para o reúso sistemático de certos artefatos como HARA, elementos arquiteturais e de erros de componentes que podem ser usados como entrada aos plugins de análise CHESS-FLA e SBA. Para resolver esses problemas, neste trabalho é proposta uma abordagem dirigida a modelos para: apoiar o uso sistemático do apoio ferramental CHESS para o projeto arquitetural e atividades de engenharia de segurança em conformidade com as diretrizes de desenvolvimento definidas em padrões de segurança, e o gerenciamento de variabilidades e o reúso de artefatos como elementos arquiteturais de sistemas, HARA e modelos de erros de componentes. A abordagem proposta foi validada em um estudo de caso realístico no domínio automotivo.

Palavras-chave: Sistemas criticos, gerenciamento de variabilidades, dependability engineering, CHESS, BVR.

Abstract

Safety-critical systems are computer systems which the occurrence of a failure may lead to catastrophic consequences, ranging from damage to the environment, financial, property damage to injuries and loss of human life. Due to the critical nature of these systems, their safety properties should be analyzed and verified at different levels of abstraction. Thus, Hazard Analysis and Risk Assessment (HARA) should be performed to identify the potential threats to the overall safety, followed by the analysis of the propagation of these threats throughout the system's architecture (e.g., using Fault Tree Analysis - FTA), and finally, the analysis of how components can contribute to the occurrence of the system-level failures, e.g., using Failure Modes and Effects Analysis (FMEA) should be done. The performance of dependability engineering activities are needed to produce artifacts required by safety standards and certifying authorities for certification and release of systems for operation. However, the production of safety/dependability engineering artifacts may cause an increase on both the project effort and costs. Reuse-oriented techniques, e.g., Software Product Line Engineering (SPLE), together with model-based techniques, have been largely adopted by the industry in the development of safety-critical systems. These techniques generally increase quality, productivity, and reduce the costs of producing design and dependability engineering artifacts. There is in the market, a series of tools that provide support to safety-critical systems engineering, e.g., MATLAB Simulink and HiP-HOPS, OSATE AADL & Error Annex, and CHESS. These tools support the specification of safety-critical systems' architectural and error models. The CHESS Toolset for example, provides support for design, dependability engineering, and code generation for safety-critical systems. Although the high level of maturity of the CHESS toolset, there is a lack of guidelines/approaches to support its systematic use in the development of safety-critical systems. Additionally, the lack of integration with SPLE tools, e.g., Base Variability Resolution (BVR), imposes barriers on the reuse of architectural and component fault models, which can be used as inputs to Failure

Logic and State-Based Dependability Analyses. In order to fill this gap, this project proposes a model-based approach to support the systematic use of the CHESS toolset for safety-critical system design and analyses in compliance with safety standards, and the systematic reuse of architectural artifacts and component fault models. The proposed approach was validated through a realistic case study from the automotive domain.

Keywords: Critical systems, variability management, dependability engineering, CHESS, BVR.

Agradecimentos

Aos meus pais e irmão pelo apoio incessável.

À todos os professores do Departamento de Ciência da Computação (DCC) da UFJF e em especial aos professores André Luiz de Oliveira (DCC/UFJF), Alexander Rasin (DePaul CDM) e Louis Oliphant (Hiram College) pelos seus ensinamentos e por todas as oportunidades oferecidas.

"All human wisdom is contained in these two words - Wait and Hope"

Alexandre Dumas (The Count of Monte Cristo)

Contents

List of Figures

List of Tables

List of Abbreviations

DCC	Departamento de Ciência da Computação
UFJF	Universidade Federal de Juiz de Fora
BVR	Base Variability Resolution
UML	Unified Modeling Language
SysML	Systems Modeling Language
HARA	Hazard Analysis and Risk Assessment
FTA	Fault Tree Analysis
FMEA	Failure Mode Effects Analysis
SIL	Safety Integrity Level
SPL	Software Product Line
SPLE	Software Product Line Engineering
BDD	Block Definition Diagram
IBD	Internal Block Diagram
FPTC	Fault Propagation and Transformation Calculus
FLA	Failure Logic Analysis
SBA	State-Based Analysis
VSpec	Variant Specification
VP	Variation Point

1 Introduction

1.1 Context

Critical systems are computer systems ranging from small devices to more complex systems such as industrial, aerospace or automotive systems. Due to their critical nature, a single failure in a system of this kind may produce catastrophic consequences (SOMMERVILLE, 2018). As a result, critical systems must satisfy availability, reliability, safety and security requirements (SOMMERVILLE, 2000).

Since these systems should address availability, reliability and safety requirements, a safety-critical system engineering process must consider both development and safety engineering activities. This is necessary in order to verify the system's safety properties at different levels of abstraction such as requirements, design, detailed design and implementation. At the requirements level, potential threats to the system safety must be identified, the risks caused by these threats must be estimated and measured, and safety requirements, must be allocated aimed at eliminating or minimizing effects of the occurrence of these threats in the overall system safety. In the design level, the system's architecture must be analyzed, in order to identify how some failures in certain components can be a threat to the system's safety. In the detailed design level, an analysis of the impact of component failures in system level and how these failures propagate throughout the system's architecture must be done. Finally, in the implementation level, it is identified how each one of the system's components can directly or indirectly contribute to system-level failures (CZERNY et al., 2010).

A set of model-driven approaches exist in the literature and offer support to both the architectural design and safety engineering activities e.g. CHESS, AADL & Error Annex, and MATLAB/Simulink/HiP-HOPS. These approaches allow the specification of systems' architectural aspects and error information in a single model. Both certifying authorities and safety standards, e.g., ISO 26262, endorse the use of model-driven approaches during the development of critical systems.

The CHESS toolset supports the engineering of critical systems by providing support for architectural modeling, which certain system architectural elements can be specified via components, and their connections (data or electrical pulses inputs/outputs). The CHESS toolset also supports component error modeling where failure probabilities (usually based on time) and failure propagation information can be attached to architectural elements that were previously specified in the model. Likewise, CHESS allows system engineers to perform a series of different analysis and simulations upon a system's model to find out more about the its behavior, or even for obtaining some evidence that the system is acceptably safe. By doing so, valuable information can be extracted from these analyses such as the probability that a system may fail during a certain period of time, or the error modes that can be raised by the system or its components under certain conditions. These results can be further used to align the system design with its previously defined safety requirements and goals (CZERNY et al., 2010).

Component-based and Software Product Line Engineering (SPLE) approaches have been widely adopted by the critical systems development industry, especially in aerospace and automotive domains. This is due to the benefits that large-scale reuse provides and the fact that safety-critical systems development companies, are constantly trying to achieve shorter delivery times and increased quality in their products (PHOL; BöCKLE; LINDEN, 2005). In this context, variations can occur on both the architectural and component error models of a safety-critical system depending on how the system was designed or due to the existence of different versions or configurations of a system that were derived from a base model. Due to the lack of reuse-oriented techniques and tools that support modeling and management of variability on system models, the BVR Tool Bundle was built (HAUGEN; ØGåRD, 2014). Thus, variability management on both architectural and error models specified in SysML/CHESS-ML can be achieved by integrating the previously mentioned BVR Tool Bundle with the CHESS toolset.

1.2 Motivation/Problem

Due to the uprising demand of products that are considered variant-rich safety-critical systems, in other words, systems that have one or more possible variants, variability management is becoming an important practice in the development of systems (VILLELA et al., 2014). Current industry practices have been trying to focus on applying software reuse techniques and prioritizing the reuse of software/hardware elements in the planning and development phases of their products (VILLELA et al., 2014). As a result, the development of such products are becoming more focused on the development of systems product lines or families of systems using the paradigm known as Systematic Software Reuse, thus, leaving old practices such as developing unique and independent systems (FRAKES; ISODA, 1994 apud VILLELA et al., 2014).

The CHESS toolset provides critical systems engineers with tools that allow the definition of system architectural models, components, subcomponents and component error models. It also supports the execution of Failure Logic, State-Based Dependability analyses and code generation. Although the CHESS toolset provides native support to a broad set of features, it does not provide full support for Systematic Software Reuse, since it does not address variability management on its models.

Currently, there is a lack of approaches strictly focused on supporting variability management on models defined using CHESS. Some tools, e.g. BVR Tool Bundle, offer support for variability management in PapyrusUML and SysML based models. Since the language used by the CHESS toolset, CHESS-ML is based on PapyrusUML's UML and SysML, it is possible to seamlessly integrate the BVR Tool Bundle with the CHESS toolset in order to support variability management on both architectural and component error models of systems specified in CHESS (JAVED; GALLINA, 2018).

Although the higher level of maturity of the CHESS toolset, there is a lack of guidelines/approaches to support its systematic use in the development of safety-critical systems. Additionally, the lack of approaches for integrating CHESS and SPLE tools, e.g., Base Variability Resolution (BVR), impose barriers on the reuse of architectural design and component fault models specified with the support of the CHESS toolset. Thus, due to the uprising demand for the specification of systems aiming at achieving the systematic reuse of components, there is a need for approaches that support variability modeling and management on both CHESS architectural and error models.

Architectural models can vary according to the targeted system configuration. As a consequence, variability can be present in various architectural elements such as ports, components and connectors among the members of a systems family. Variation can also be found in a components error models. A single component can have a certain error model depending on the chosen system variant. A component can behave in a certain way when used in a system variant but it can also have a completely different behavior if implemented in a different system variant.

1.3 Objectives

In order to supply the lack of approaches that support the systematic use of the CHESS toolset and variability modeling and management on both architectural and error models developed using the CHESS toolset, this project proposes a model-based approach to support the systematic use of the CHESS toolset for safety-critical system design and dependability analysis in compliance with safety standards, and the systematic reuse of design and component fault models. The proposed approach was validated through a realistic wheel braking system case study from automotive domain.

1.3.1 General Objective

Propose a model-driven approach to support the systematic use of the CHESS toolset, CHESS-FLA, CHESS-SBA and a software product line engineering based approach to support the reuse of both CHESS generated design and component error model assets.

1.4 Work Structure

This capstone paper is organized into an introduction and five chapters. In **Chapter 2**, a literature review covering all the important concepts to help the reader understand the research contributions presented in Chapters 3 and 4 are introduced. **Chapter 3** presents a systematic approach to support both the architectural and error model specification of safety critical systems using the CHESS toolset as well a set of steps to execute and interpret the results of two different types of analyses generated by the CHESS toolset. The proposed approach was extracted from (BRESSAN et al., 2018), and was validated in a detailed and realistic case study using a real-life automotive safety critical system. The presented case study is an slightly modified version of the one presented in the previously cited work. **Chapter 4** presents a systematic approach to support the reuse of CHESS design and component error model related assets. The proposed approach was validated in a case study using the same real-life safety critical system presented in Chapter 3. **Chapter 5** lists and describes a set of related works. Finally, **Chapter 6** presents a summary of this work's contributions, their benefits and limitations, and a brief description of the future work that will be developed from what has been proposed in this project.

2 Literature Review

This chapter presents the concepts and terms that were used as a theoretical background throughout this project, which are required to better understand the research contributions presented in the Chapters 3 and 4.

2.1 Safety Engineering

Failures within a system can happen at any time no matter how well tested the given system was. Failures can cause a series of consequences ranging from no effect to catastrophic severity. The occurrence of failure can lead to financial loss, environmental disasters and even put human lives in danger. System failures might happen due a set of reasons such as *bad design*, *lack of testing*, *lack of maintenance*, *use of low quality hardware components* and/or *human error* (VERMA; AJIT; KARANKI, 2015).

System engineers are strongly advised to apply *safety engineering* activities during the development of safety-critical systems. Performing safety engineering activities are mandatory/recommended by both certifying authorities and safety standards to achieve the safety certification and release the system for operation. By doing so, engineers can ensure that the system under development be acceptably safe, e.g., by reducing the probability and the number of possible failures that can emerge from the system when under operation. Activities related to safety engineering are intended to identify potential threats and the risk that they pose to the overall system safety, quantify performance and, based on the obtained results, allocate safety requirements to reduce the probability of occurrence and the number of possible threats that a system may suffer. These activities may also help on increasing the system's performance and looking for better ways to eliminate or minimize the effects of the occurrence of potential safety threats that were previously identified (VERMA; AJIT; KARANKI, 2015).

Existing safety standards, e.g., ISO 26262 for the automotive and SAE ARP 4754A for the aerospace domains, provide guidance for the Safety Engineering life-cycle of safety-critical systems. Such guidance is presented in the form of a development methodology and establishes desirable quality attributes to be addressed to achieve safety certification. There are two examples of safety standards that are vastly used by the critical systems industry: the ISO 26262 that supports the development and certifies automotive systems and the SAE ARP 4754A that supports the development and provides certification to aerospace systems.

2.1.1 Safety Terminology

Safety properties of critical systems should be analyzed and verified at different levels of abstraction. This is done in order to certify that a given system addresses reliability, availability, safety and security requirements. Thus, safety engineering activities should be performed to identify the hazards, failures and harms that a given system may produce due to a malfunctioning of a hardware or software component (CZERNY et al., 2010).

According to Papadoupoulos e McDermid (1999), **safety** can be defined as "freedom from unacceptable risk" and it is only associated with hazardous failures while reliability, relates to all potential failures (LEVESON, 1986). **Reliability** is defined as the probability a given component, under a certain time interval and environmental conditions, will continuously perform its intended function (LEVESON, 1995).

Hazards are potential sources of harm caused by a malfunctioning system or set of systems that implement a function (ISO, 2011). According to MoD (2007), a **system** is a combination of elements such as materials, tools, equipment, and software in a operating environment that should perform a given task or achieve a specific purpose. Additionally, a **safety risk** is defined as the the severity and likelihood of a harm. A **harm** is defined by MoD (2007) as "death, physical, injury, damage to the health of people, or damage to property or to the environment". A **failure**, according RTCA (2012) is "the inability of a system or system component to perform a required function within their specified limits". A **fault**, still according to RTCA (2012), is the "manifestation of an error".

Safety critical systems should address a set of safety requirements or risk reduction measures. These measures should be applied in order to reduce the effects of potential faults associated with failures and hazards (MOD, 2007). Safety requirements can be divided into three categories: i) system-safety requirements which are allocated to system-level hazards, ii) functional safety or derived safety requirements which are responsible for minimizing or eliminating the effects, in system safety, of a failure (ISO, 2011) and iii) safety integrity requirements, responsible for specifying, in terms of severity and probability, the risks associated with failures, hazards and functional safety requirements (OLIVEIRA, 2016).

Assurance is the the provision of evidence that a given product satisfies its safety requirements through planned and systematic actions (RTCA, 2012) and generated evidence. Assurance is the information that serves as the starting-point for the generation of (safety) arguments. Based on how true these claims are, arguments can be established, challenged and contextualized (SUN, 2013).

2.2 Safety Life-Cycle

A safety life-cycle comprises a set of steps and activities aimed at achieving overall functional safety in a documented manner, through the design, specification and implementation of safety instrumented systems. It involves all known development phases starting from the concept phase up to the decommissioning of the project, when all the Safety Instrumented Functions are no longer available (ALI, 2005).

Each one of the main known safety standards, e.g., ISO 26262 and the SAE ARP 4754A have its own safety life-cycle being each one, adapted to the domain which each standard addresses. Even though these life cycles can differ from each other in some aspects, they do share some common design, dependability and safety engineering activities among each other within its phases such as allocation of safety-related responsibilities among people or within entire departments or organizations, system design, item integration and testing, Hazard Analysis and Risk Assessment or HARA, allocation of safety requirements, component fault analysis, Fault Tree Analysis (FTA) and Failure Modes Effects Analysis (FMEA).

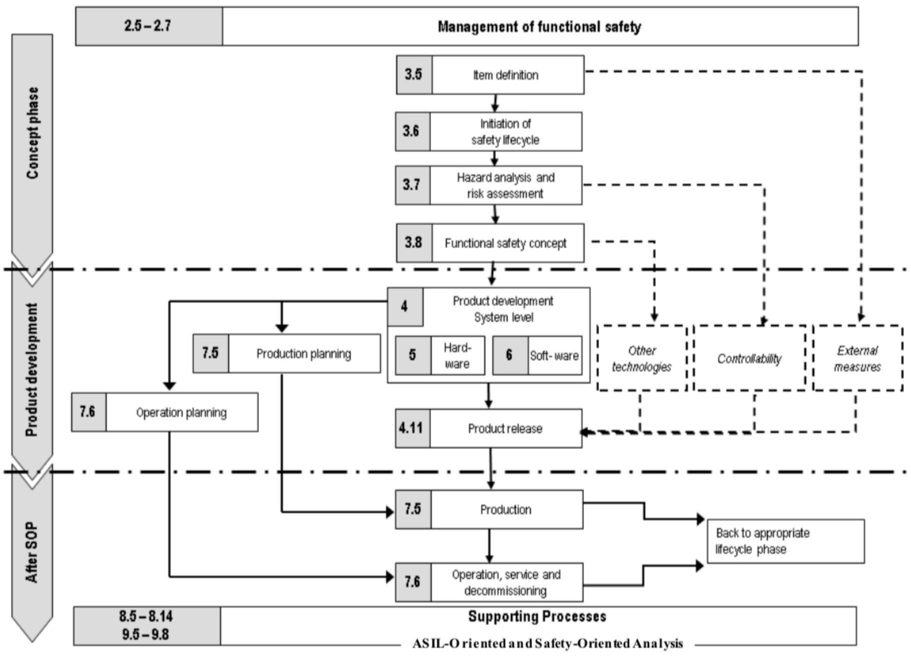

2.5 – 2.7
Management of functional safety
Concept phase
3.5
Item definition
3.6
Initiation of safety lifecycle
3.7
Hazard analysis and risk assessment
3.8
Functional safety concept
Product development
4
Product development System level
7.5
Production planning
5
Hard-ware
6
Soft-ware
Other technologies
Controllability
External measures
7.6
Operation planning
4.11
Product release
After SOP
7.5
Production
Back to appropriate lifecycle phase
7.6
Operation, service and decommissioning
8.5 – 8.14
9.5 – 9.8
Supporting Processes
ASIL-Oriented and Safety-Oriented Analysis

Figure 2.1: The ISO 26262 safety life cycle, its phases and activities (ISO, 2011)

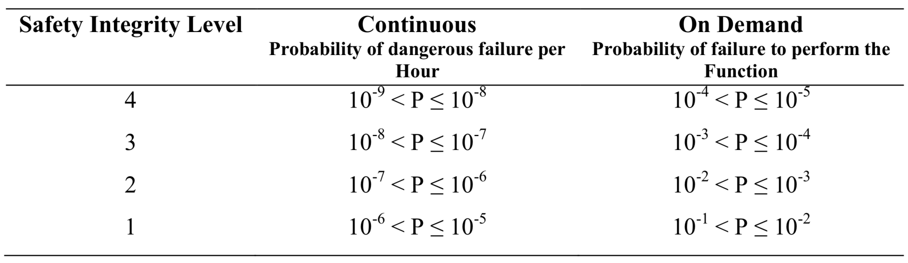
2.2.1 HARA and Allocation of Safety Requirements

HARA is an acronym for Hazard Analysis and Risk Assessment. The Hazard Analysis process can be divided into five main steps: the identification of specific actions or tasks that are necessary in order to achieve the defined project goals, the discovery of hazards associated with these tasks, the identification of all the risks linked to the identified hazards, the development of a checklist containing all the steps necessary to eliminate or minimize the risks and the creation of a working procedure where ways of completing each task the safest way possible will be described (MTU, 2018).

Risk assessment is done after identifying hazards and other factors that may cause some kind of harm to the system, its users or the environment and the process is responsible for evaluating and analyzing the risk associated with each identified hazard (CCOHS, 2018).

The allocation of safety requirements can then be done by taking all the information produced by the risk assessment phase and using it to determine ways, procedures or functionality that would help eliminating the known hazards or at least minimizing its effects in case the hazard can't be eliminated at all as safety integrity or functional safety requirements (CCOHS, 2018) (OLIVEIRA, 2016).

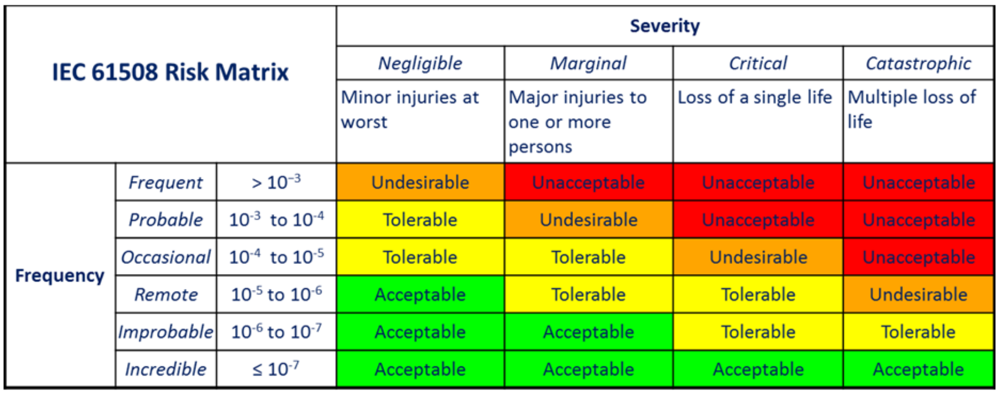
Most safety standards such as the ISO 26262 and the IEC 61508 define their safety integrity requirements according to probabilistic or quantitative criteria where they can be defined taking into consideration factors such probabilities, fault-tree operation, unavailability and mean-time-to-failure (LITTLEWOOD; STRIGINI, 1993 apud OLIVEIRA, 2016). The IEC 61508 standard for example, determines its safety integrity requirements taking into account two different factors. The first one is the probability that a failure may happen during the execution of continuous or high demand functions and the second one, the probability that an on demand system function might fail when requested or executed. The standard defines different System Integrity Levels according to the two previously mentioned probability criteria and the possible variation ranges within them where the SIL 4 is the most stringent level and the SIL 1 the least one (OLIVEIRA, 2016):



Safety Integrity Level	**Continuous** **Probability of dangerous failure per Hour**	**On Demand** **Probability of failure to perform the Function**
4	$10^{-9} < P \leq 10^{-8}$	$10^{-4} < P \leq 10^{-5}$
3	$10^{-8} < P \leq 10^{-7}$	$10^{-3} < P \leq 10^{-4}$
2	$10^{-7} < P \leq 10^{-6}$	$10^{-2} < P \leq 10^{-3}$
1	$10^{-6} < P \leq 10^{-5}$	$10^{-1} < P \leq 10^{-2}$

Figure 2.2: IEC 61508's Safety Integrity Levels (IEC, 2010)

Risk matrices are considered a very helpful resource and guidance to the allocation of safety requirements to system components. In the IEC 61508 Risk Matrix, it is not only possible to determine if a certain risk is acceptable, tolerable, undesirable or unacceptable by combining failure frequency and severity together but also, to use it to allocate specific safety requirements to certain components that will help decreasing failure frequency, failure severity or both so risks can be minimized or even mitigated within the system or component in question:



IEC 61508 Risk Matrix			Severity			
			Negligible	*Marginal*	*Critical*	*Catastrophic*
			Minor injuries at worst	Major injuries to one or more persons	Loss of a single life	Multiple loss of life
Frequency	*Frequent*	$> 10^{-3}$	Undesirable	Unacceptable	Unacceptable	Unacceptable
	Probable	10^{-3} to 10^{-4}	Tolerable	Undesirable	Unacceptable	Unacceptable
	Occasional	10^{-4} to 10^{-5}	Tolerable	Tolerable	Undesirable	Unacceptable
	Remote	10^{-5} to 10^{-6}	Acceptable	Tolerable	Tolerable	Undesirable
	Improbable	10^{-6} to 10^{-7}	Acceptable	Acceptable	Tolerable	Tolerable
	Incredible	$\leq 10^{-7}$	Acceptable	Acceptable	Acceptable	Acceptable

Figure 2.3: IEC 61508's Risk Matrix (IEC, 2010)

2.2.2 FTA and FMEA

Fault tree analysis and FMEA support the safety analysis in a system by identifying potential faults, ever since the early stages of a system's development process. Both techniques are broadly used by the industry in the development of safety-critical systems of various different domains such as the automotive, aerospace and nuclear power domains (OLIVEIRA, 2016).

A fault tree provides important information about undesirable events and its consequences to the system that is being analyzed (NASA, 2002). These events can be associated with hardware/software failures and human errors. The information expressed in a fault tree is specified through a series of logical gates connecting events. An output (top) event, can be connected or related to one or many other input (lower) events. The tree, through its gates and connectors, represents the events needed for the occurrence of a higher event (VERMA; AJIT; KARANKI, 2015).

Figure 2.4 contains an example of a fault tree from the Fault Tree Handbook (NRC, 1981) describing the steps necessary for the occurrence of a 'rupture of the pressure tank after the start of pumping' within a pressure control system. According to the schematics shown in the fault tree, the rupture can occur by a 'tank rupture due to improper installation', 'tank rupture (primary failure)' or 'tank rupture (secondary failure)'. The 'tank rupture (secondary failure)' can be triggered by '***ruptures due to internal over-pressure*** caused by continuous pump operation for t > 60 sec or "secondary failure from other out-of tolerance conditions (eg. mechanical thermal)" (OLIVEIRA, 2016).

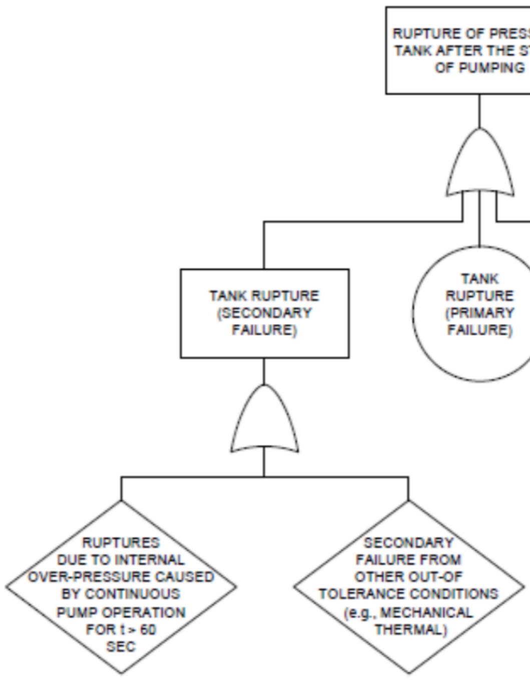

RUPTURE OF PRESSURE TANK AFTER THE START OF PUMPING
TANK RUPTURE (SECONDARY FAILURE)
TANK RUPTURE (PRIMARY FAILURE)
TANK RUPTURES DUE TO IMPROPER INSTALLATION
RUPTURES DUE TO INTERNAL OVER-PRESSURE CAUSED BY CONTINUOUS PUMP OPERATION FOR t > 60 SEC
SECONDARY FAILURE FROM OTHER OUT-OF TOLERANCE CONDITIONS (e.g., MECHANICAL THERMAL)

Figure 2.4: Rupture of the pressure tank fault tree (NRC, 1981)

According to Verma, Ajit e Karanki (2015), ”Fault tree analysis is a failure oriented, deductive and top-down approach, which considers an undesirable event associated with the system as the top event, the various possible combinations of fault events leading to the top event are represented with logic gates”.

FMEA is a systematic method where remedial actions should be taken considering its results in order to mitigate the identified failures in a more effective way. FMEA results are composed by a system’s rank of all potential failure modes in terms of their criticality levels. The FMEA aims on analyzing the system-wide effects of failures of certain components or items and classify potential failures according to their identified severity (VERMA; AJIT; KARANKI, 2015).

2.3 Safety Standards and Certification

Safety standards provides guidance for developing safety-critical systems according to the targeted safety objectives established per level of integrity. The guidance can be in the form of activities to be performed and artifacts to be produced to achieve a given safety objective. Thus, such guidance can be optional or mandatory depending on the targeted level of integrity. Standards help engineers to verify if a given system addresses the safety requirements (HARPERCOLLINS, 2018). Due the level of maturity achieved by model-based techniques, safety standards, e.g., ISO 26262 for automotive, and SAE ARP 4754A for avionics, recommend their usage to support both development and safety assessment activities.

The ISO 26262 comes from an adaptation of the IEC 61508 standard that has a more general nature and certifies systems of all different domains available in the industry. One of its main objectives is to supply some of the more specific automotive systems certification necessities. The ISO 26262 was developed by a partnership of a set of automotive and research companies such as BMW, Bosch, Toyota and IBM and defines a functional life-cycle for products of the automotive domain. By dong so, it is applied in every single safety life-cycle activity of critical systems, incorporating electrical, electronic or software components. The standard supports only a single class of automotive systems being those terrestrial systems that weight no more than 3.5 tons and that are designed for a general purpose and not for an specific purpose such as vehicles that are adapted to people with disabilities, for example (CZERNY et al., 2010).

2.4 Model-Driven Engineering

Model-driven development techniques are approaches of great importance in the development of critical systems. A model can defined as a simplified representation of a process (GABRY, 2017). In critical systems engineering, models are used to describe system requirements and attributes in a non-ambiguous or semi-ambiguous manner. Another very important characteristic supported by some models is the possibility of generating artifacts from them. Models can be transformed into other models or even executable code eg. Generation of SQL code through an entity-relation model diagram transformation.

Model-Drivel Engineering or MDE, according to Kleiner, Albert e Bézivin (2009), ”is a research area that considers the main software artifacts as graphs”. MDE was conceived due to the need that organizations have to maintain themselves homogeneous in a way that different software systems artifacts can be easily combined or separated from each other.

Models that are used to aid MDE professionals in their activities can be divided into three different levels: the terminal model, the metamodel and the metametamodel (M1, M2, M3). Terminal models are models that can be used to describe a certain system's characteristics. These models are expressed through modeling languages such as UML or SysML by capturing the characteristics of system in question and expressing them through these languages. A metamodel is the abstract syntax, expressed through a model, of a modeling language. A metametamodel captures a metamodeling language that is used to define metamodels, conceptual foundation (FLEUREY, 2006 apud NASTOV, 2013).

Companies are highly encouraged to use MDE in order to improve the short-term and long-term productivity of their developers and professionals. By doing so, MDE aims to reduce the development team's effort on developing a new piece of software and increase the organization's profits (AKTINSON; KUHNE, 2003 apud NASTOV, 2013).

2.5 Dependability

Dependability is the ability an item or system has to operate as it is supposed to whenever required or throughout a determined time interval. It can also be described as the 'time-related quality characteristics of an item' (IEC, 2015).

Dependability can be quantitatively measured through certain attributes or qualities such as availability, reliability, recoverability, maintainability, maintenance support performance, durability, safety and security. According to IEC (2015), the availability of an item is the ability this item has to "be in a state to perform as required". Reliability is the ability an item has to perform completely free of failures under certain conditions for a certain time interval. Recoverability, as the name says, is the ability of an item to recover from a failure without the need of a corrective maintenance (IEC, 2015).

Avionics systems for example, should be available and perform as they're supposed to throughout an entire flight. Redundant systems should also be available throughout the entire duration of the flight in case some important system fails. The availability of redundant systems is an important feature when it comes to reducing recoverability rates since the faster the switch between a faulty system and a healthy system, the shorter the needed recoverability time is.

2.6 Model-Based Safety Analysis

As stated earlier in this work, there is in the literature, a vast support for model-driven development and analysis techniques. Among these techniques, there are component based techniques which both the architectural project and activities related to safety engineering are supported. According to (OLIVEIRA, 2016), ”Compositional safety analysis techniques provide formal and semi-formal languages to support the specification, composition, and analysis of the system failure behavior based on safety information about the system components”.

Component based techniques are implemented by various known tools that are frequently used by professionals in the development of critical systems such as CHESS, AADL, Matlab Simulink and HiP-HOPS. The CHESS toolset for instance, implements the State-based Dependability Analysis. This type of analysis aids developers on measuring how available and reliable a certain system is and building or reshaping their systems in order to make them safer and to ensure that they attend its safety requirements, based on the results obtained through the analysis.

Formal verification techniques are used to generate results from model simulations. Tools that use formal verification techniques such as Altarica 3.0 (BATTEUX et al., 2013), simulate normal system operation and determine fault effects to system safety by injecting them in the system that is being simulated (PAPADOPOULOS et al., 2011 apud OLIVEIRA, 2016).

2.7 The CHESS Toolset

CHESS is a collection of tools available as an Eclipse IDE extension that support the specification of systems, its architectural features and error models. The MDT Papyrus extension is used alongside CHESS to support the creation of graphical models in UML/SysML/CHESS-ML. The CHESS toolset supports all the steps defined in a system development process. Such steps involve some practices such as the requirements definition, systems’ functional architecture, physical and logical modeling, software design and its applications in hardware components. CHESS also supports automatic ADA code generation (INTECS, 2016).

The CHESS toolset also supports many different types of analyses. One of the main kinds of analysis supported by CHESS is the Failure Logic Analysis (FLA) where given certain input error modes for a component's or system's input ports, the tool propagates these values throughout the system's components or components' subcomponents in accordance to the error model (specified using FPTC rules) defined for each system or component and returns as a final result, the error modes generated by the analyzed system and all its components or components and all its subcomponent's output ports or in other words, if an error mode such as an omission or a *valueSubtle* will be outputted by the system's or component's output ports given a certain combination of error modes as an input. It basically tells if the system or component in question, will mitigate, propagate or output a different error modes from the ones that were set as input error modes for that system or component.

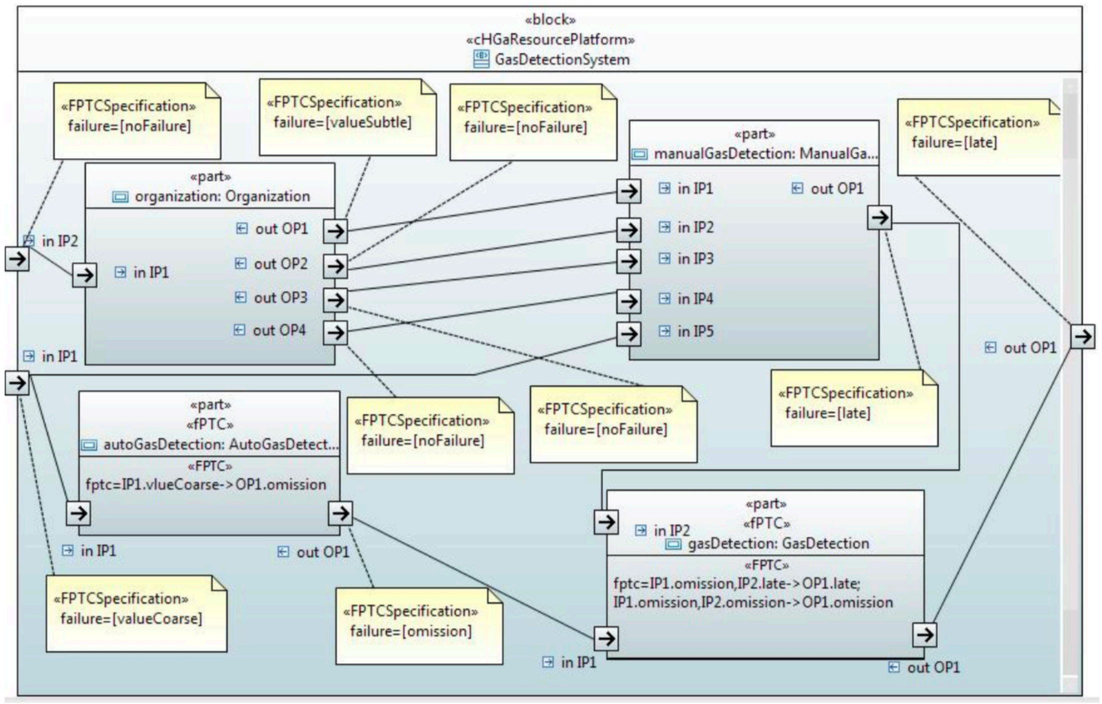

«block»
«cHGaResourcePlatform»
GasDetectionSystem
«FPTCSpecification» failure=[noFailure]
«FPTCSpecification» failure=[valueSubtle]
«FPTCSpecification» failure=[noFailure]
«FPTCSpecification» failure=[late]
«part» organization: Organization
in IP2
in IP1
out OP1
out OP2
out OP3
out OP4
«part» manualGasDetection: ManualGa...
in IP1
out OP1
in IP2
in IP3
in IP4
in IP5
out OP1
in IP1
«part» «fPTC» autoGasDetection: AutoGasDetect...
«FPTC» fptc=IP1.vlueCoarse->OP1.omission
«FPTCSpecification» failure=[noFailure]
«FPTCSpecification» failure=[noFailure]
«FPTCSpecification» failure=[late]
in IP1
out OP1
«part» «fPTC» gasDetection: GasDetection
in IP2
«FPTC» fptc=IP1.omission,IP2.late->OP1.late; IP1.omission,IP2.omission->OP1.omission
«FPTCSpecification» failure=[valueCoarse]
«FPTCSpecification» failure=[omission]
in IP1
out OP1

Figure 2.5: A CHESS model containing back propagated Failure Logic Analysis results (SEFER, 2015)

A second type of analysis that is supported by the CHESS toolset is called State-Based Dependability Analysis. This type of analysis is capable of calculating and return-ing the probability of a system remaining healthy or free of errors from instant 0 until a given time (Reliability), the probability of a system being on a healthy state on a defined instant (Availability/Instantaneous) or the fraction of time that a system can keep itself free of errors during a certain given time frame or interval (Availability/Interval of Time) taking into account, just like in the previously mentioned Failure Logic Analysis, the architectural and error models of the system that is being analyzed (UNIFI, 2017).

The analysis results are back-propagated by CHESS into the specified model and, through these obtained results, critical systems engineers can modify or restructure the model in such a way that the rebuilt system will satisfy certain previously established safety requirements or safety goals or hit desirable levels of Availability and Reliability (INTECS, 2016).

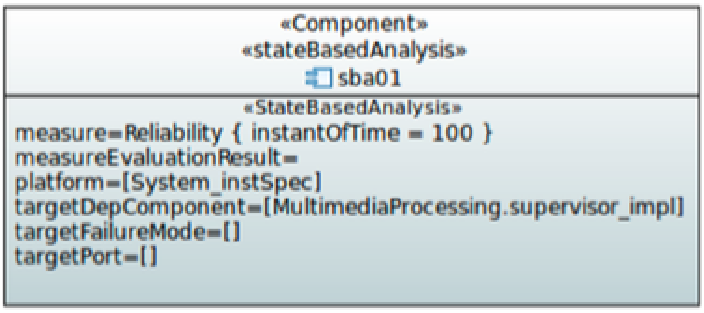

«Component»
«stateBasedAnalysis»
sba01
«StateBasedAnalysis»
measure=Reliability { instantOfTime = 100 }
measureEvaluationResult=
platform=[System_instSpec]
targetDepComponent=[MultimediaProcessing.supervisor_impl]
targetFailureMode=[]
targetPort=[]

Figure 2.6: State-based Dependability Analysis results are back propagated into the 'measureEvaluationResult' parameter of the «stateBasedAnalysis» stereotyped «Component» (UNIFI, 2017)

In addition to offering the previously mentioned features, the CHESS toolset also offers an Automotive Profile to support the development of automotive systems. The Automotive Profile introduces certain properties such as inheritance and ASIL decomposition verification and certain elements that can be used in a system's modeling process that must fit the demands listed by the ISO 26262 (INTECS, 2016).

2.7.1 Component Fault Modeling

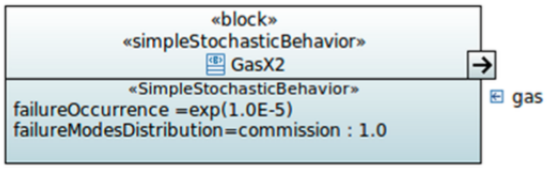
The CHESS toolset implements three different types of component fault models: «SimpleStochasticBehavior», «FLABehavior» and «ErrorModelBehavior». These stereotypes and its elements are used to specify components' error behavior characteristics such as fault propagation, probability of occurrence of an internal fault and repair times. This information can be also further used to execute dependability analyses.

Components stereotyped as «SimpleStochasticBehavior» are components that are only affected by one type of internal faults, generate one kind of errors and are capable of generating different types of failure modes with a certain probability. The «SimpleStochasticBehavior» stereotype contains three different parameters that can be used to describe components' dependability information: through the 'faultOccurrence' parameter, engineers can specify as a time distribution, the average time to the occurrence of an internal fault. The 'failureModesDistribution' is an optional parameter that describes all the possible output failure modes as well as their probabilities and uses the following grammar:

```
<FMD> ::= <D> | <PD> | <PD>; <PD>
<PD> ::= <PORT> <D>
<D> ::= { <FP> }
<FP> ::= <F> : <P> | <F> : <P>; <FP>
      <F> is a failure mode, <P> is a probability value,
      <PORT> is a port of the component
```

Figure 2.7: The 'failureModesDistribution' grammar (UNIFI, 2017)

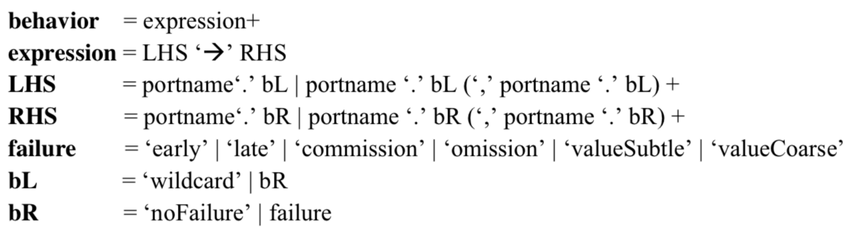
Finally, the "repairDelay" parameter can be used to specify, as a time distribution, the time a component needs to be repaired or to recover from an internal failure. Figure 2.8 shows a «SimpleStochasticBehavior» stereotyped component which fails once in every 1.0E5 hours and whenever it fails, it outputs a commission failure mode through the 'gas' output port:



failureOccurrence =exp(1.0E-5)
failureModesDistribution=commission : 1.0

Figure 2.8: A «SimpleStochasticBehavior» stereotyped «Component» (UNIFI, 2017)

The second stereotype supported by the CHESS toolset, «FLABehavior» can be used to specify components' failure logic behavior. It is used to express how components propagate, mitigate and transform failure modes experienced on their input ports through a technique called FPTC. Different behaviors of individual components can be expressed by a set of logical expressions called FPTC rules and these behaviors can be separated into four different groups: source, sink, propagational and transformational behaviors. A source behavior happens when a component outputs a certain failure mode due to an internal fault within it, a sink behavior happens when a component detects and mitigates an error mode propagated through its input, a proportional behavior happens when an error mode on a component's input is propagated through its output and a transformation behavior is when a component outputs a different kind of failure mode compared to the one received on its input (SLJIVO et al., 2016). The syntax used to specify FPTC rules within models generated by the CHESS toolset is shown on Figure 2.9:



behavior = expression+
expression = LHS '→' RHS
LHS = portname'.' bL | portname '.' bL (',' portname '.' bL) +
RHS = portname'.' bR | portname '.' bR (',' portname '.' bR) +
failure = 'early' | 'late' | 'commission' | 'omission' | 'valueSubtle' | 'valueCoarse'
bL = 'wildcard' | bR
bR = 'noFailure' | failure

Figure 2.9: The FPTC grammar supported in the CHESS toolset by CHESS-FLA (SEFER, 2015), (SLJIVO et al., 2016)

As shown above on the FPTC grammar, there are five different failure modes supported by it and those failure modes can be separated into three different groups (SEFER, 2015):

- Value related failures:
  - *valueSubtle*: this failure mode denotes that an input or output has deviated from its expected range of values in a way which humans cannot detect it (GALLINA; PUNNEKKAT, 2011 apud SEFER, 2015).
  - *valueCoarse*: this failure mode denotes that an input or output has deviated from its expected range of values in a detectable way by humans (GALLINA; PUNNEKKAT, 2011 apud SEFER, 2015).
- Time related failures:

  - *early*: this failure mode happens when input or output is provided earlier than expected (SEFER, 2015).
  - *late*: this failure mode happens when input or output is provided later than expected (SEFER, 2015).

- Provision related failures:
  - *omission*: this failure mode happens when an input or output is omitted (SEFER, 2015).
  - *commission*: this failure mode happens when input or output is provided when not expected (SEFER, 2015).

The *noFailure* keyword denotes that the input or output in question had not received any kind of failure modes and the *wildcard* keyword is used to indicate that a certain input port can receive any kind of the previously mentioned failure modes and still generate a certain output failure mode e.g. The *in1.wildcard* → *out1.omission* rule indicates that the component in question will always produce an omission failure mode through its out1 output port no matter what failure mode it experiences through its *in1* input port (SLJIVO et al., 2016).

The third stereotype supported by the toolset, the «ErrorModelBehavior» stereotype can be used to provide a more detailed error behavior model by supporting the specification of more technicalities related to faults, errors and failure modes of system elements. This stereotype uses state machine diagrams stereotyped as «ErrorModel» containing all the information related to certain component's fault behavior. An state machine comprises the following elements (UNIFI, 2017):

- Initial state: represents a component's 'healthy' state.
- Error states: represents an error state using an UML State stereotyped as «ErrorState». Eg. undetected.
- Internal faults: connects the initial state to an error state through an UML transition stereotyped as «InternalFault». Through this type of transition, the time to a fault occurrence can be specified using the occurrence parameter.

- Internal propagation: UML transitions stereotyped as «InternalPropagation» where the time after which the propagation occur (delay), relative probability of occurrence (weight) and the indication if an external fault will be produced by it (externalFaults) can be specified through the stereotype parameters.

- Failures: represents the failure modes propagated to the component's output ports through the mode parameter. It is represented by an UML «Failure» stereotyped transition.

The «ErrorModel» state machine diagram on Figure 2.10 shows a «ErrorModel-Behavior» component's error behavior. The diagram indicates that the component fails once in every 10^{-4} hours/days/years with an internal fault. Once it fails, there is a 50% chance it will go to a "LateDetection" error state and produce a delayed output through its 'out' output port and there is also a 50% chance it will go to an Undetected error state and omit whatever was supposed to be outputted by its 'out' output port:

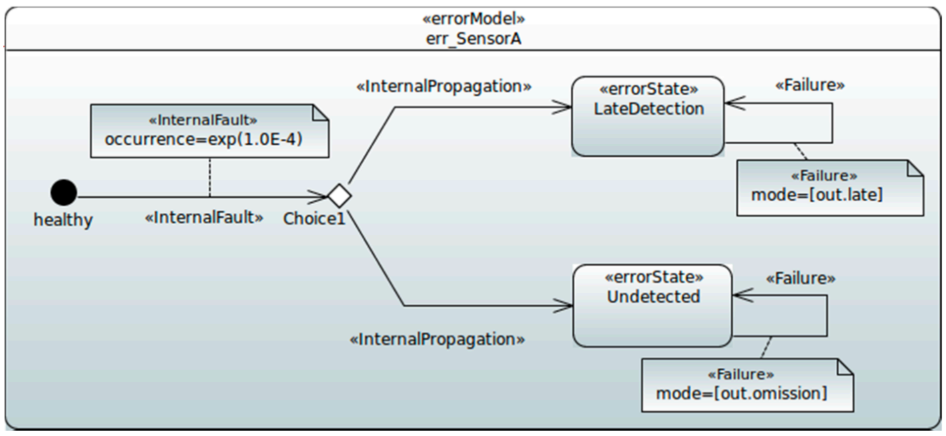


Figure 2.10: An «errorModel» used to describe an «ErrorModelBehavior» stereotyped component's error behavior (UNIFI, 2017)

2.8 Software Product Line Engineering (SPLE)

Software product line engineering (SPLE) is the application of Product line engineering activities during the development of software families. Software families are a set of systems that share a common set of features among each other. These activities are mainly aimed in the reduction of development costs, the provision of customized products at reasonable costs, reduction of time to market and product quality enhancement (PHOL; BöCKLE; LINDEN, 2005).

The Software Product Line Engineering approach comprises the processes: Domain and Application Engineering. In the Domain Engineering, both commonality and variability analysis of a product family is performed. Thus, defining the reusable platform in which all types of software artifacts, e.g., requirements, design, realization and tests can be found (PHOL; BöCKLE; LINDEN, 2005). The Application Engineering process, is responsible for deriving SPL products from the platform established in the domain engineering". During application engineering, an SPL product is derived from a set of specific artifacts previously defined in the domain engineering phase (PHOL; BöCKLE; LINDEN, 2005).

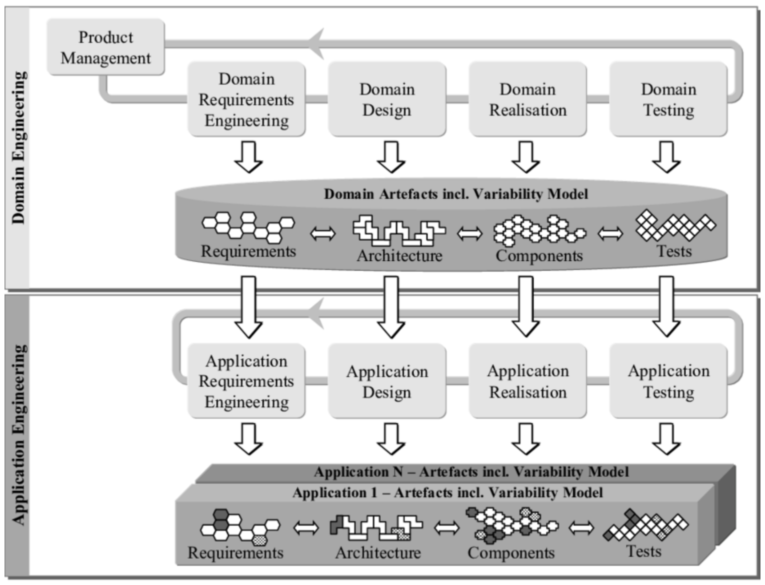


Figure 2.11: The Product Line Engineering Process (PHOL; BöCKLE; LINDEN, 2005)

2.9 Product Line Variability Metamodel

A variability metamodel comprises two key concepts: ***variation point*** and ***variant***. A *variation point* is a place where variability can occur. A *variant*, on the other hand, is something that provides an instance for domain artifacts, specified in a *variation point*, that can vary (PHOL; BöCKLE; LINDEN, 2005 apud OLIVEIRA, 2016). *Excludes* constraints indicating that the instance of a *variation point* is restricted to the instance of another and *requires* constraints indicate that a *variation point* requires the instance of a *variant* belonging to a distinct *variation point*, can be used to define different types of inter-dependencies between *variation points* and *variants* (OLIVEIRA, 2016). Figure 2.12 shows the variability metamodel and all the relations between *variants* and *variation points*:

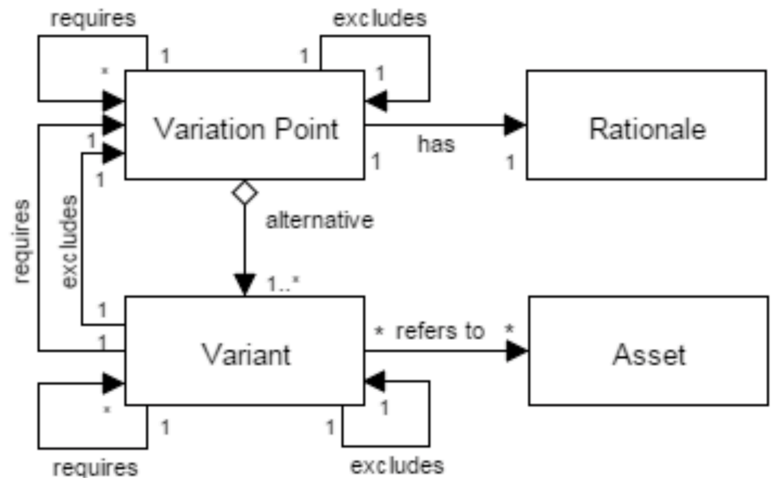


Figure 2.12: The variability metamodel defined by (BACHMANN et al., 2004)

Variations in a system's both the architectural and error models can produce different effects when it comes to hazard causes and the safety requirements allocated to prevent those hazards effects. Certain variation points must be chosen depending on the selected relevant operation modes and their criticalities in the context model. As previously mentioned in the beginning of this paragraph, these selections can impact directly on how an specific design can contribute to system level hazards previously identified in the product's feature model (HABLI; KELLY; PAIGE, 2009 apud OLIVEIRA, 2016).

2.10 Reuse in Safety Engineering and Software Product Lines

Software Reuse is defined as the usage of existing system's development artifacts or characteristics in the inception of new system's. In other words, the use of certain functionality or aspects such as source code fragments, specifications, design structures or even documentation from other existing projects in the development of a newer system (FREEMAN, 1983 apud KRUEGER, 1992).

Software Product Lines are defined as a set of similar systems that are designed through similar means and developed from a series of common artifacts and characteristics. Software Product Lines are created and consequently maintained, tested or improved through a series of existing practices defined by Software Product Lines Engineering concepts (BIGLEVER, 2013). Software Reuse is one of the main concepts addressed by Software Product Lines once these are strongly based in the reuse or the share of components and other aspects of systems belonging to a product line among themselves.

Due to the benefits provided by the large scale reuse, especially in the development of software systems and systems that must align with certain safety standards such as ISO 26262 or the SAE ARP 4754A, the industry has started adopting reuse practices in safety-critical system's development processes (OLIVEIRA et al., 2018). This is due the adoption of large-scale reuse may impact on the reduction of the time-to-market, and increase on the quality of the produced artifacts such as dependability.

2.11 BVR and the BVR Tool Bundle

As mentioned previously, due to the industry's increasing demand for practices supported by Software Product Line Engineering especially when aimed towards the development of critical systems product lines, the BVR language or Base Variability Resolution was conceived. BVR was created to support variability management in EMF compliant models belonging to the safety domain (HAUGEN; ØGåRD, 2014).

A Software Product Line can be modeled using a feature tree represented in a diagram where certain system's features or functionality are classified as mandatory, optional or alternative. For example, a certain car can have many different types of engines, and each different engine choice may impact on the possibility of this car having an automatic transmission or not.

The BVR Tool Bundle provides a set of plug-ins for the Eclipse platform that support BVR language, and the use of certain practices such as variability management and derivation of models defined in UML/SysML through the MDT Papyrus plug-in. The BVR tool bundle also supports feature modeling, and resolution modeling which allows the definition of different configurations that a certain system may have (SINTEF, 2018).

2.12 Variability Modeling and Management

The BVR tool bundle implements a set of features that allows engineers to specify software product lines. In order to do so, engineers must follow a series of steps to successfully model, manage software product lines and generate derived models using the tool.

The fist step consists on modeling the family features using BVR's VSpec Editor. Figure 2.13 shows a feature tree for a diesel car and each one of its VSpecs in which its gear box can be either automatic and manual. If the selected box type is automatic, the car will have AWD traction and have a 140 hp engine. Otherwise, the car will be either AWD or FWD being the AWD version either 140 hp or 110 hp and the FWD version being exclusively 110 hp.

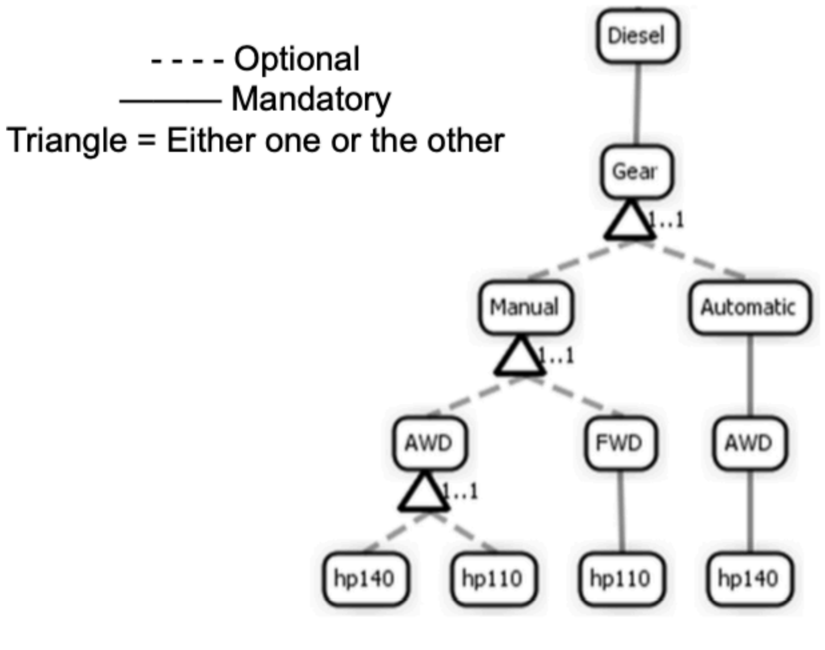
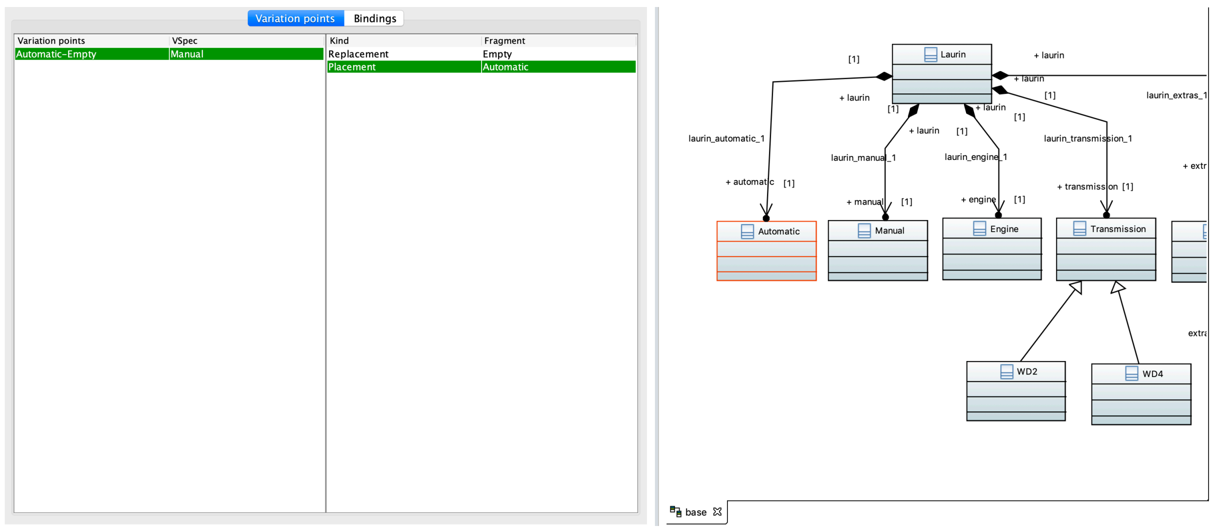


Figure 2.13: A Diesel car's VSpec tree (HAUGEN; ØGåRD, 2014)

After defining the system's VSpecs, Variants should be specified by selecting components within the model as placements and/or replacements. For each pair of placement and replacement engineers should also specify to which previously specified VSpec each one of them refers. Components inside a placement will be taken away from the model while components inside a replacement will stay in the final derived model. Figure 2.14

shows a variant containing a placement and a replacement, linked to the 'Manual' VSpec. In this example, the car's automatic gear box was set as a placement and will be taken out of the final derived model if the 'Manual' VSpec is set to True in the resolution model.



Variation points	VSpec
Automatic-Empty	Manual

Kind	Fragment
Replacement	Empty
Placement	Automatic

Figure 2.14: A variation point containing a placement and replacement linked to the 'Manual' VSpec

The third step consists on setting each previously defined VSpec as 'True' or 'False' according to the product variant in question. After doing so, the engineer can then execute BVR upon the desired Papyrus model. Once BVR is done executing, a new derived Papyrus model will be generated.

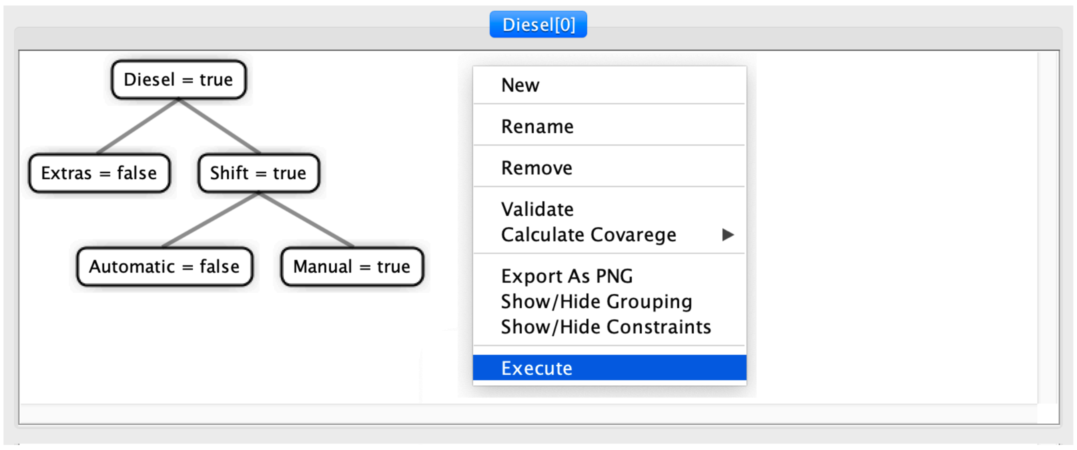


Figure 2.15: A simplified diesel car's VSpec tree on the Resolution editor

3 The Proposed CHESS Model-Based Design and Dependability Analysis Process

In this chapter, the proposal of a model-based to support the systematic usage of the CHESS toolset in the engineering of safety-critical systems, is presented. The proposed approach was previously published as a conference paper by Bressan et al. (2018), and it provides guidance to support engineers on specifying both architectural and error models using the CHESS toolset. This chapter also provides guidance for generating safety evidence, recommended by standards, e.g., ISO 26262, through the different types of analysis tools bundled within the toolset.

The proposed approach was specified in SPEM 2.0 activity diagrams. SPEM 2.0 is a modeling notation for specifying software processes. SPEM 2.0 comprises a set of model elements such as activities, tasks, and milestones. An activity represents a process that can be decomposed into tasks. A Task or Task Definition element defines a work unit which cannot be decomposed. A Milestone element represents a significant achievement within a project (OMG, 2008). As illustrated in Figure 3.1, the proposed approach comprises four main phases: Architectural/System Model Design, System Error Modeling, Failure Logic and State-Based Dependability Analyses.

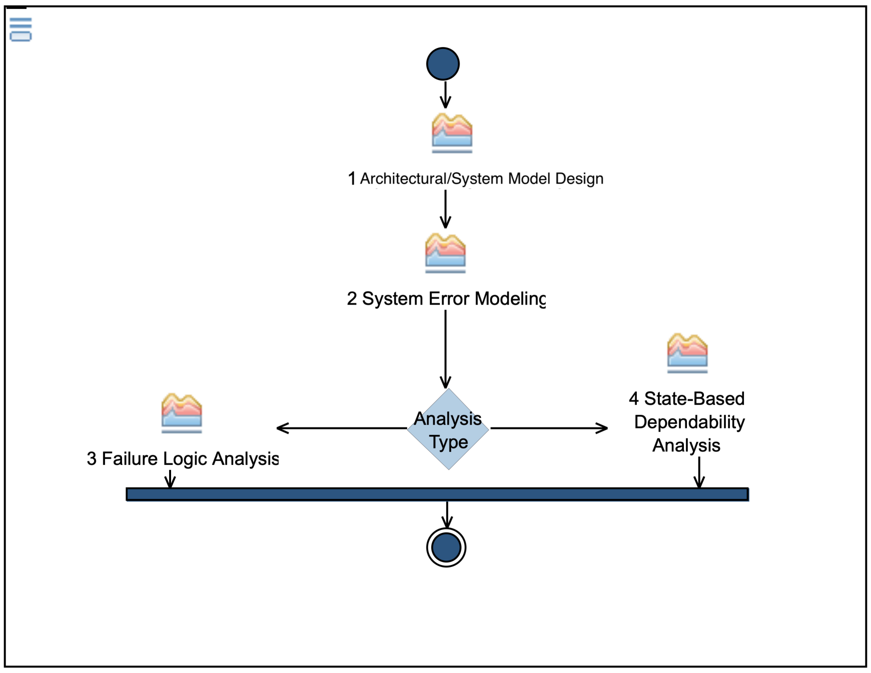

1 Architectural/System Model Design
2 System Error Modeling
Analysis Type
3 Failure Logic Analysis
4 State-Based Dependability Analysis

Figure 3.1: CHESS model-based design and dependability analysis phases

The Hybrid Braking System (HBS) is be used through this report to illustrate the application of proposed process and its steps. HBS is a real world automotive braking system originally designed on MATLAB/Simulink meant to be used with electric road vehicles that integrate one electric motor per wheel (CASTRO; ARAUJO; FREITAS, 2011 apud BRESSAN et al., 2018). As shown on Figure 3.2, the braking system comprises 4 subsystems, 24 subsystem components, 6 components and 69 different connections among subsystems and sub-components. The combined action of the electrical In-Wheel Motors (IWMs) with the Electromechanical Brakes (EMBs) provides braking power to the system. Since the IWMs work as generators when the brakes are activated transforming the vehicle's kinetic energy into electrical energy to feed the Powertrain Battery, increased vehicle autonomy can be achieved by using the brakes. Since HBS should always be available to the driver to guarantee their safety, the braking torque generated by it should always be correct and it can never be omitted otherwise the driver could suffer catastrophic consequences in case the system malfunctions (BRESSAN et al., 2018).

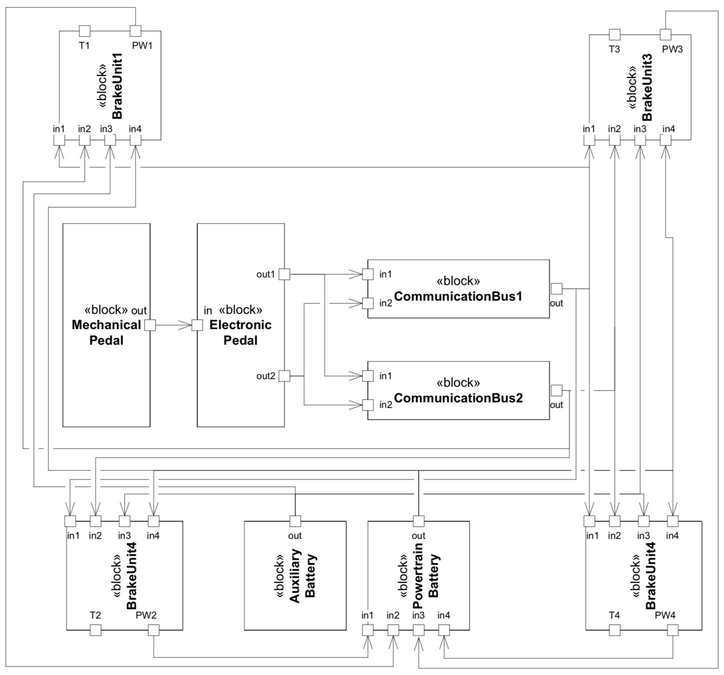


Figure 3.2: HBS' Architectural Model (OLIVEIRA, 2016)

3.1 Architectural/System Design

Input: The requirements document.

Output: The system's architectural model containing the system «block»s or components and their ports specified in a *Block Definition Diagram* (BDD). The system's and its internal subcomponents internal architectures (instances of «block»s as «part»s, and the data connections between them) described through an Internal Block Diagram (IBD).

The 'Architectural/System Design' phase prescribes a set of steps to define a system's architectural model. In order to specify a model using the CHESS toolset, a CHESS project and a Papyrus UML model should be created. Latter, a new UML *Block Definition Diagram* should be created within CHESS's modelSystemView. All the system's components, sub-components and their input and output ports must be specified in this diagram using SysUML «block» and «port» elements (BRESSAN et al., 2018). Figure 3.3 shows the required tasks and activities to successfully produce a CHESS system architectural model.

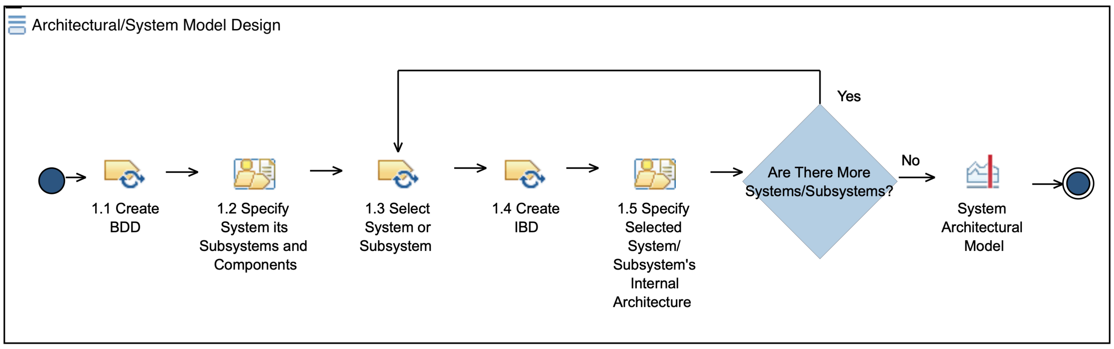

Architectural/System Model Design
1.1 Create BDD
1.2 Specify System its Subsystems and Components
1.3 Select System or Subsystem
1.4 Create IBD
1.5 Specify Selected System/ Subsystem's Internal Architecture
Are There More Systems/Subsystems?
Yes
No
System Architectural Model

Figure 3.3: CHESS model-based design and dependability analysis process and its phases

Figure 3.4 shows the HBS SysML Internal Block Diagram. HBS is a real world automotive braking system originally designed in MATLAB/Simulink. HBS is meant for integration in electrical vehicles, in particular for propulsion architectures that integrate one electrical motor per wheel [9]. The term hybrid comes from the fact that braking is achieved throughout the combined action of the electrical In-Wheel Motors (IWMs), and the frictional Electromechanical Brakes (EMBs). One of the most important features of this system is that the integration of IWM in the braking process allows an increase in the vehicle's range. Thus, while braking, IWMs work as generators and transform the vehicle kinetic energy into electrical energy that is fed into the Powertrain Battery. HBS should not raise omission of braking torque or incorrect value of braking torque failures in the wheel's while braking, since the occurrence of such hazardous events can lead to catastrophic consequences for the driver

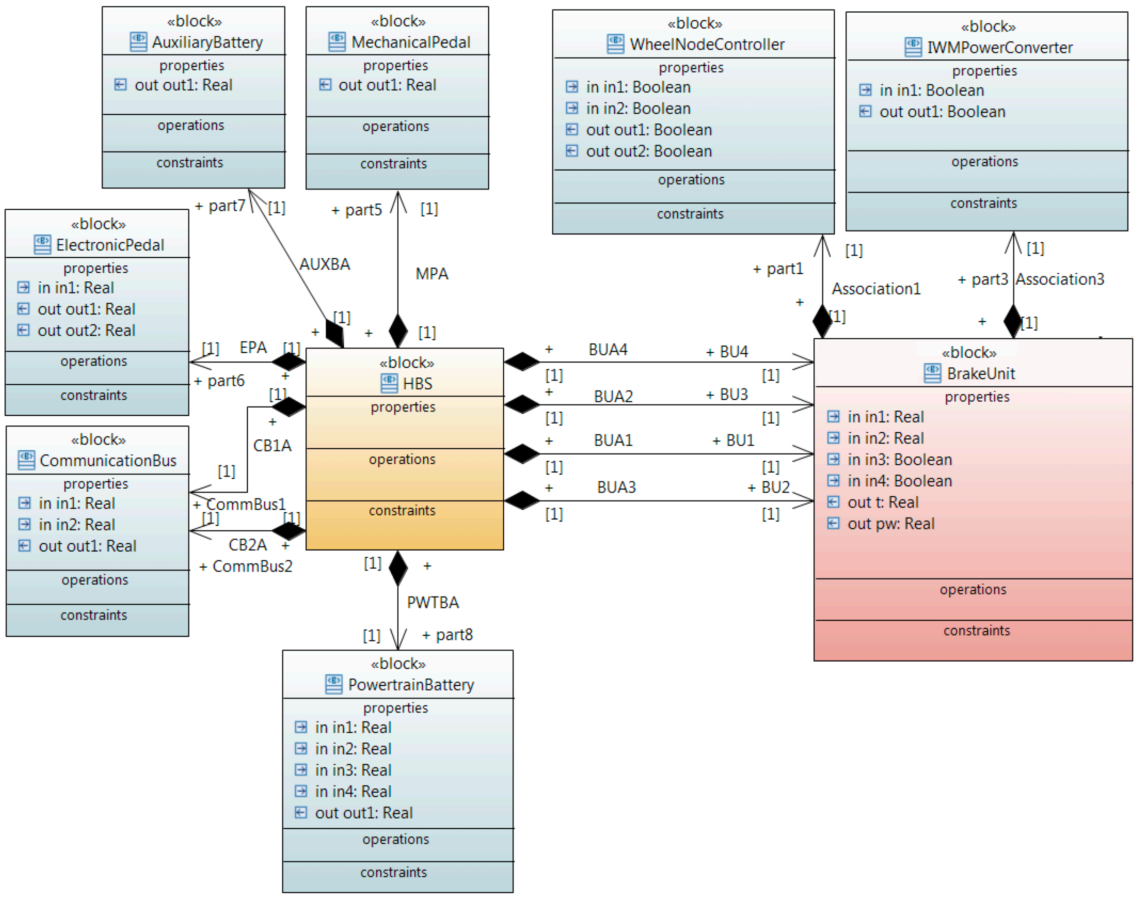


Figure 3.4: The HBS system and a set of its components and subcomponents on a Block Definition Diagram

Once all systems, components and subsystems have been specified specified, system's and subsystem's internal architectures should be defined in UML Internal Block Definition diagrams. In CHESS, subsystem's architectures are specified using instances of elements previously defined on a Block Definition Diagram, its ports, and by determining links or connections between them. The HBS comprises four Wheel Brake Unit subsystems, so a total of 5 different Internal Block Diagrams must be created in order to specify the HBS system and all of its subsystems' architectures (BRESSAN et al., 2018).

Additionally, since all elements that represent components and subsystems within a system are instances of the SysML «block»s specified in a Block Definition Diagram, these elements can be instantiated multiple times within a given system. This allows engineers to provide different safety information for different instances or «part»s of the same SysML «block». Thus, the number of SysML Internal Block Diagrams and «block» elements needed can be decreased when compared to the model presented by Bressan et al.

(2018). The number of Internal Block Diagrams can be reduced to 2 since a single Brake Unit SysML «block» can be created and instantiated 4 times within HBS. The HBS also comprises two Communication Buses components, specified as an «block» in the Block Definition Diagram, and then instantiated twice on HBS' Internal Block Diagram.

Figure 3.5 illustrates the Internal Block Diagram for the Brake Unit component. A wheel Brake Unit comprises 6 internal components. The Wheel Node Controller (WNC) component calculates the amount of breaking that is going to be produced by the brake unit, and it send commands to both the EMB PowerConverter (Electromechanical-Brake-Power-Converter) and IWM Power Converter (In-Wheel-Motor-Power-Converter) components. These commands further propagated to both EMB(Electromechanical-Brake) and IWM (In-Wheel-Motor) components. The IWM decreases the vehicle's kinetic energy by converting into energy. The EMB is used alongside the IWM to provide enough braking power since it can be fairly decreased on high speeds when HBS' Powertrain Battery is close to being charged. The Add component outputs both the torque generated and the total amount of power generated by the brake (BRESSAN et al., 2018):

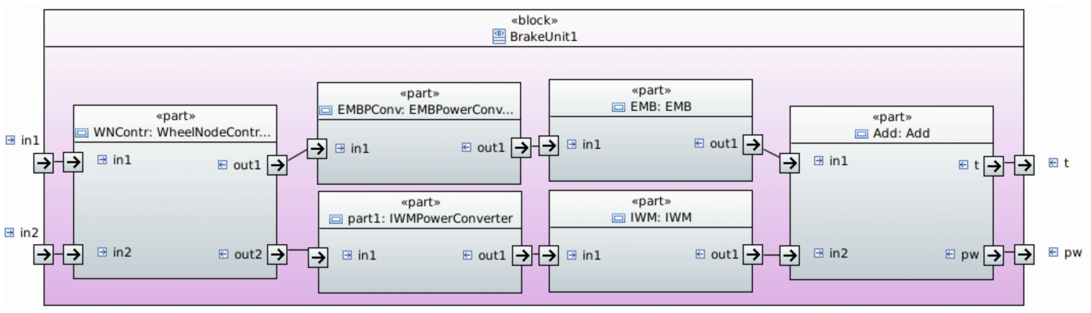


Figure 3.5: Internal Block Diagram containing the Brake Unit subsystem architecture (BRESSAN et al., 2018)

3.2 Error Model Specification and Dependability Analysis

The process presented in the previous section defines a phase that covers system's error modeling activities, and two distinct phases for each one of the two different types of analyses supported by the CHESS toolset. In the 'System Error Modeling Phase', components are selected and annotated with dependability or failure propagation information, depending on the error stereotypes applied to each one.

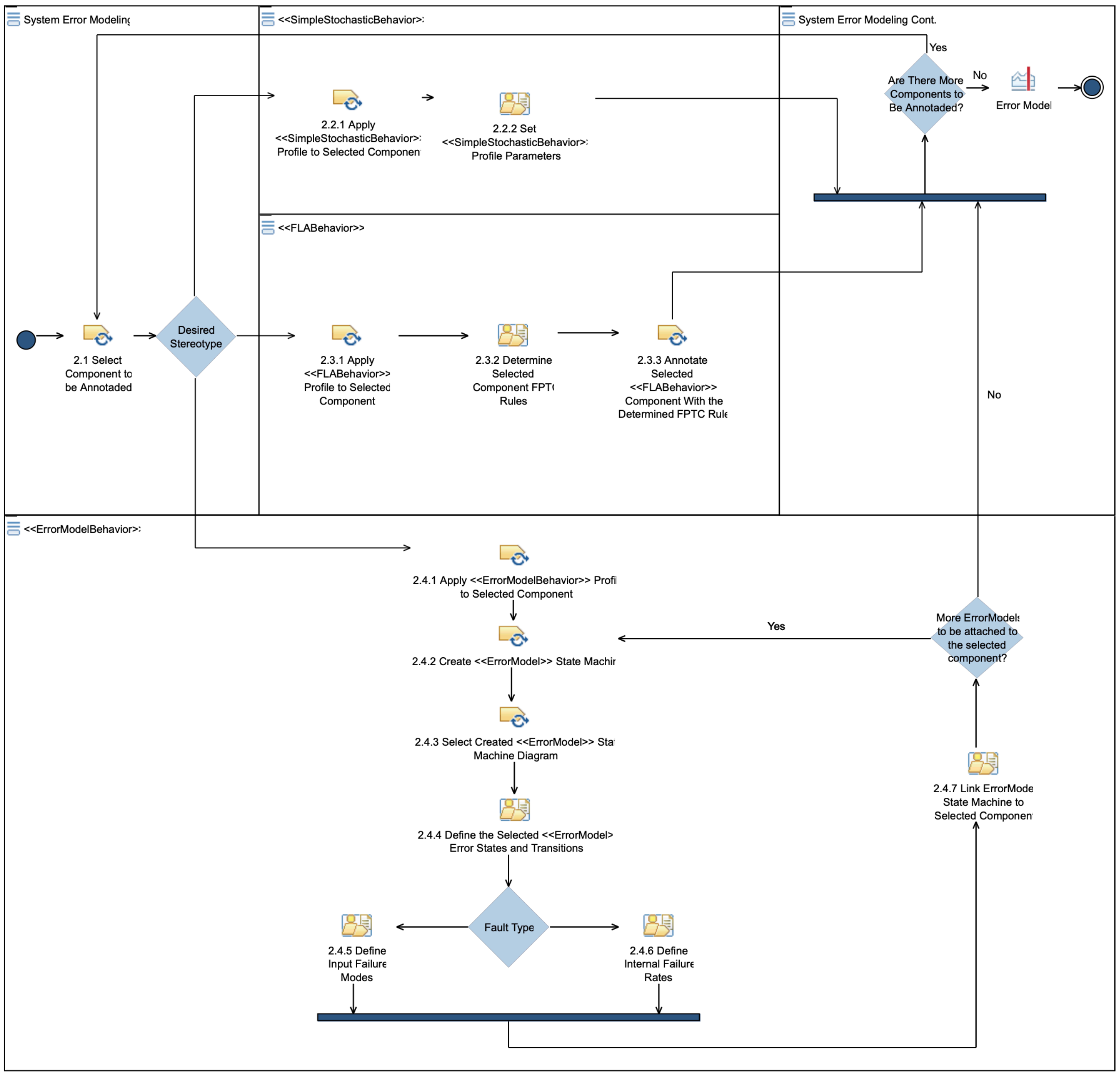

System Error Modeling
<<SimpleStochasticBehavior>>
System Error Modeling Cont.
Yes
Are There More Components to Be Annotaded?
No
Error Model
2.2.1 Apply <<SimpleStochasticBehavior>> Profile to Selected Component
2.2.2 Set <<SimpleStochasticBehavior>> Profile Parameters
<<FLABehavior>>
2.1 Select Component to be Annotaded
Desired Stereotype
2.3.1 Apply <<FLABehavior>> Profile to Selected Component
2.3.2 Determine Selected Component FPTC Rules
2.3.3 Annotate Selected <<FLABehavior>> Component With the Determined FPTC Rule
No
<<ErrorModelBehavior>>
2.4.1 Apply <<ErrorModelBehavior>> Profile to Selected Component
2.4.2 Create <<ErrorModel>> State Machine
Yes
More ErrorModels to be attached to the selected component?
2.4.3 Select Created <<ErrorModel>> State Machine Diagram
2.4.7 Link ErrorModel State Machine to Selected Component
2.4.4 Define the Selected <<ErrorModel>> Error States and Transitions
2.4.5 Define Input Failure Modes
Fault Type
2.4.6 Define Internal Failure Rates

Figure 3.6: Set of tasks and activities to enrich components with dependability or failure propagation information in CHESS

The error information of a given component can be specified in three different ways. Different stereotypes or CHESS-ML profiles, e.g., «simpleStochasticBehavior», «flaBehavior» and/or «ErrorModelBehavior», can be applied to components to enrich them with error/failure information that can be further used by CHESS' analysis tools. The process illustrated in Figure 3.6, defines a set of steps to be performed when using each one of these profiles on the specification of failure behaviors of system or subsystem components: «flaBehavior» stereotyped components should be annotated with their respective FPTC rules taking into account how the component behaves when receiving certain failure modes, hardware components annotated with «simpleStochasticBehavior» must have the profile's parameters specified, and components stereotyped with the «ErrorModelBehavior» should have one or more ErrorModel state machines attached to them. Each state machine can either express how and how often an internal failure occurs or how certain failure modes are propagated throughout the component.

The following subsections describe how Failure Logic and State-Based Dependability analyses can be performed through the CHESS-FLA and CHESS-SBA plugins. Hybrid Braking System is used again to exemplify the steps and the results that can be obtained from each type of analysis.

3.2.1 Failure Logic Analysis (FLA)

Inputs: Architectural model containing components stereotyped as «FLABehavior» and annotated with error mode propagation information.

Output: FLA Analysis results are back propagated into the selected system. Error modes coming through the system's input ports are propagated through the system's components according to their FPTC rules. These components' output ports are then annotated with comments containing the error modes they outputted according to the input error modes received through their inputs.

As mentioned previously on section 2.10, CHESS' FLA Analysis propagates failure modes received through a component's or system's input through its elements annotated with failure propagation information and stereotyped as «flaBehavior» using the FPTC language and returns the failure modes the system will output. By doing this, FLA Analysis determines if a certain error mode will be mitigated, propagated or changed when outputted by the component or system being analyzed.

In order to illustrate the use of CHESS-FLA, and the execution of FLA Analysis, a Brake Unit subsystem was taken out of context, and its components were annotated with FPTC rules as shown on Table 3.1:

Table 3.1: Brake Unit components and their corresponding FPTC rules

Component	FPTC Rules
WheelNodeController	in1.omission,in2.wildcard → $out1.omission, out2.omission$;
	in1.wildcard,in2.omission → $out1.omission, out2.omission$;
	in1.valueSubtle,in2.noFailure → $out1.valueSubtle, out2.valueSubtle$;
	in1.noFailure,in2.valueSubtle → $out1.valueSubtle, out2.valueSubtle$;
	in1.valueSubtle,in2.valueSubtle → $out1.valueSubtle, out2.valueSubtle$;
	in1.valueCoarse,in2.noFailure → $out1.noFailure, out2.noFailure$;
	in1.noFailure,in2.valueCoarse → $out1.noFailure, out2.noFailure$;
	in1.valueCoarse,in2.valueCoarse → $out1.noFailure, out2.noFailure$;
EMBPowerConverter	in1.omission → $out1.omission$;
	in1.valueSubtle → $out1.valueSubtle$;
IWMPowerConverter	in1.omission → $out1.omission$;
	in1.valueSubtle → $out1.valueSubtle$;
EMB	in1.omission → $out1.omission$;
	in1.valueSubtle → $out1.valueSubtle$;
IWM	in1.omission → $out1.omission$;
	in1.valueSubtle → $out1.valueSubtle$;
Add	in1.omission,in2.wildcard → $pw.omission, t.omission$;
	in1.wildcard,in2.omission → $pw.omission, t.omission$;
	in1.valueSubtle,in2.noFailure → $pw.valueSubtle, t.valueSubtle$;
	in1.noFailure,in2.valueSubtle → $pw.valueSubtle, t.valueSubtle$;
	in1.valueSubtle,in2.valueSubtle → $pw.valueSubtle, t.valueSubtle$;

According to its FPTC rules, the *wheelNodeController* component for example, propagates an omission if any of its inputs fails with an omission, a *valueSubtle* error mode if one input fails with this mode and the other doesn't fail or both inputs fail with a *valueSublte* error mode and mitigates *valueCoarse* errors received through one input when the other doesn't fail or through both of its input ports since the *valueCoarse* component can detect and consequently mitigate *valueCoarse* errors.

Once all the system components were properly annotated with their corresponding FPTC rules, the Failure Logic Analysis can be executed by selecting the system or subsystem that is going to be analyzed and annotate the selected system's input ports with incoming error modes. CHESS-FLA can be now executed and once done running, each system component's output ports are annotated with with their respective failure mode taking into account both the system component's FPTC rules and its previously defined incoming error modes:

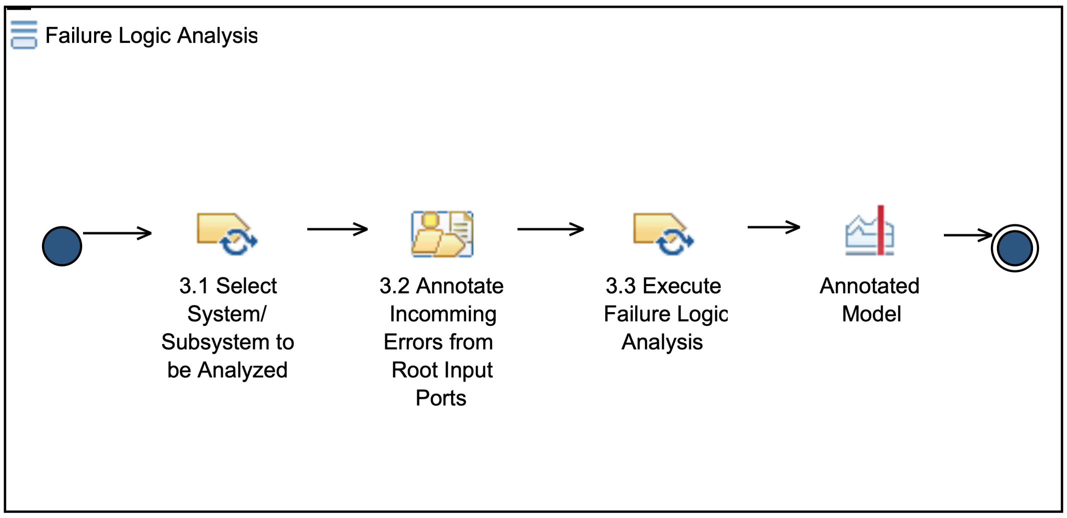

Failure Logic Analysis
3.1 Select System/ Subsystem to be Analyzed
3.2 Annotate Incomming Errors from Root Input Ports
3.3 Execute Failure Logic Analysis
Annotated Model

Figure 3.7: The set of tasks and activities required to perform the Failure Logic Analysis using CHESS-FLA

In order to exemplify how CHESS-FLA's Analysis works and back-propagates its results into the model, two different scenarios were considered. In the first scenario, the *BrakeUnit* component receives no failures through its 'in1' input port and a *valueCoarse* error mode through its *in2* port. The analysis shows that the *valueCoarse* error mode is mitigated by the *valueCoarse* thence preventing the *BrakeUnit* component from failing:

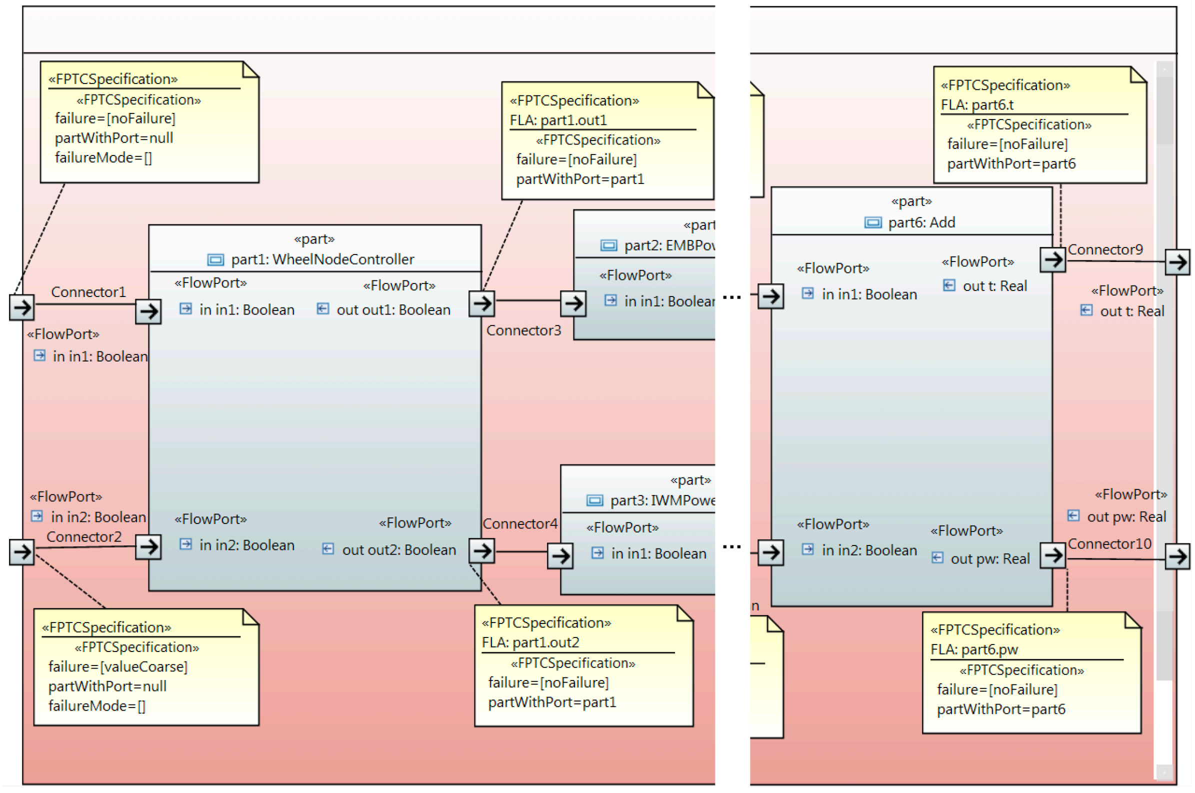


Figure 3.8: FLA Analysis results for the first scenario

The second scenario considers that the *BrakeUnit* component receives an *omission* failure through its *in1* input port, and a *valueSubtle* error mode through its *in2* input port. The FLA Analysis results for this case show that the *valueCoarse* ignores the *valueSubtle* error mode received through its *in2* port and simply propagates the omission through both its outputs. The omission is then propagated through all the other *BrakeUnit*'s components and the Add component omits both the braking torque (t) and the generated power (pw):

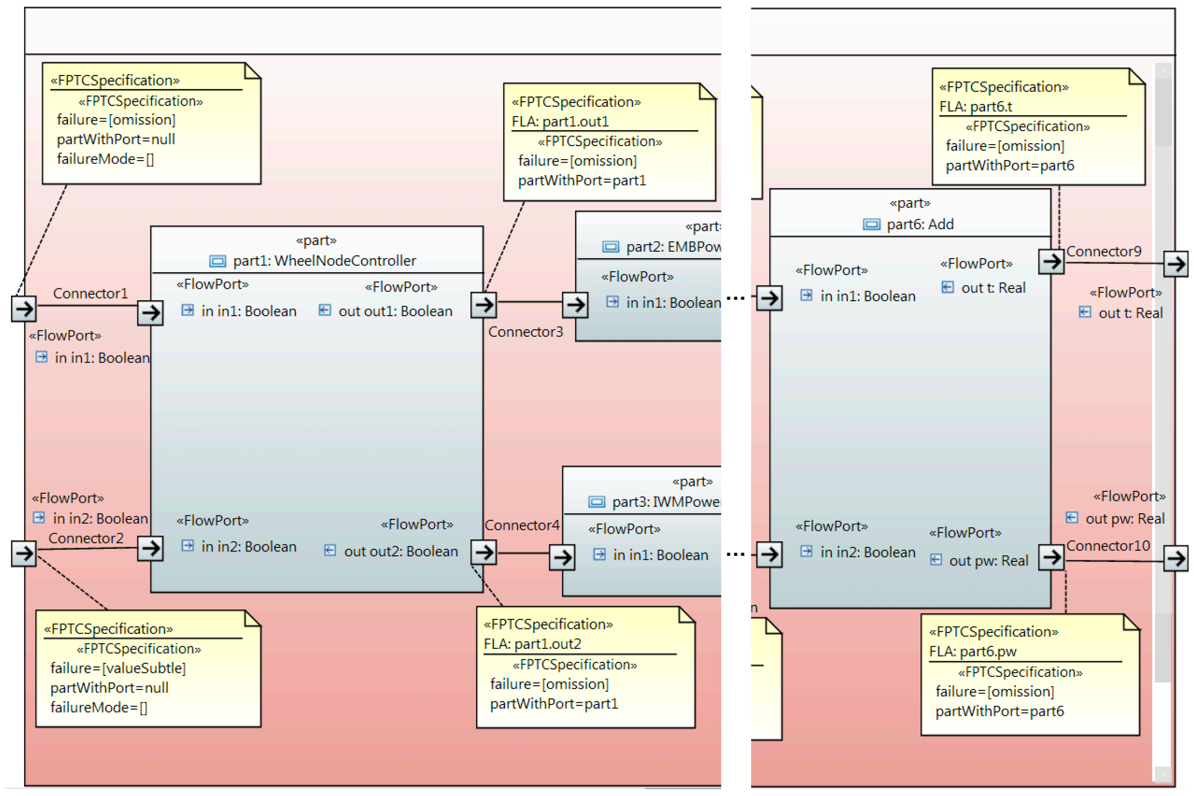


Figure 3.9: FLA Analysis results for the second scenario

3.2.2 State-Based Dependability Analysis

Inputs: CHESS architectural model components annotated with «simpleStochasticBehavior», «FLABehavior» or «ErrorModelBehavior» stereotypes, i.e., annotated with dependability or error mode propagation information.

Outputs: State-Based Dependability Analysis results back propagated into the model as a probabilistic distribution.

State-Based analysis provides a way to quantitatively evaluate the dependability attributes of a system. It uses information within CHESS specified architectural and error models to return the probability that the system or component being analyzed does not fail at a given instant (Availability) or the probability that a system remains healthy until a predefined time instant (UNIFI, 2017).

The CHESS State-Based Dependability analysis can be performed by: creating a new UML Class Diagram under «DependabilityAnalysisView», creating a «StateBasedAnalysis» stereotyped «Component» in this class, and by executing the CHESS-SBA's

State-Based Analysis:

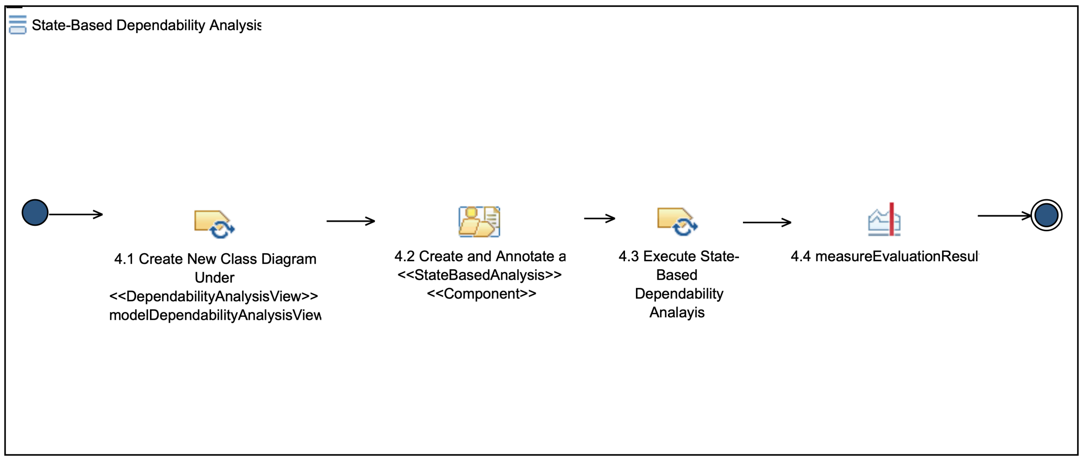

State-Based Dependability Analysis
4.1 Create New Class Diagram Under <<DependabilityAnalysisView>> modelDependabilityAnalysisView
4.2 Create and Annotate a <<StateBasedAnalysis>> <<Component>>
4.3 Execute State-Based Dependability Analayis
4.4 measureEvaluationResul

Figure 3.10: The series of tasks and activities required to perform the State-Based Dependability Analysis using CHESS-SBA

Hardware components that are part of HBS have their error models specified using the «SimpleStochasticBehavior» stereotype since these can only suffer from internal faults for not having input ports and can generate one or more different output error modes once they fail.

Software components can be annotated with both «FLABehavior» or «ErrorModelBehavior» error model stereotypes. «FLABehavior» should be used on components that propagate *failure modes* without suffering from *internal faults*. «ErrorModelBehavior» can be used on components that propagate failure modes without suffering from internal faults, and on components that can suffer from internal faults and by consequence, output an error mode through their output ports. All components that are a part from a *BrakeUnit* subsystem, except the Add component, can be annotated with «ErrorModelBehavior» to perform the CHESS-SBA. The *Add* component is annotated with «FLABehavior» stereotype, since this component does not suffer from internal faults. All other software components from the HBS which might raise internal failures were annotated with the «ErrorModelBehavior» stereotype, and their error models were specified with «ErrorModel» stereotype referencing a given component state machine.

After executing the CHESS state-based analysis, the analysis results were back propagated through the model, more specifically, to the «Component» that holds the analysis parameters. In this specific scenario, the reliability of HBS was tested where the system fails only when one of its *BrakeUnit*s omits braking torque (hence the ”*target-FailureMode*” parameter set to ”omission”). The time interval that the system is being analyzed is 8760 hours or a year. As a result, the analysis returned that the Hybrid Braking System has a 99.6% probability of continuously staying in a healthy state during a whole year.

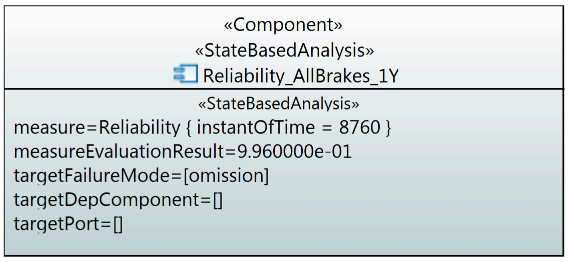

«Component»
«StateBasedAnalysis»
Reliability_AllBrakes_1Y
«StateBasedAnalysis»
measure=Reliability { instantOfTime = 8760 }
measureEvaluationResult=9.960000e-01
targetFailureMode=[omission]
targetDepComponent=[]
targetPort=[]

Figure 3.11: State-Based Analysis parameters and results

3.3 The Relation Between the Proposed Approach and the ISO 26262 Standard

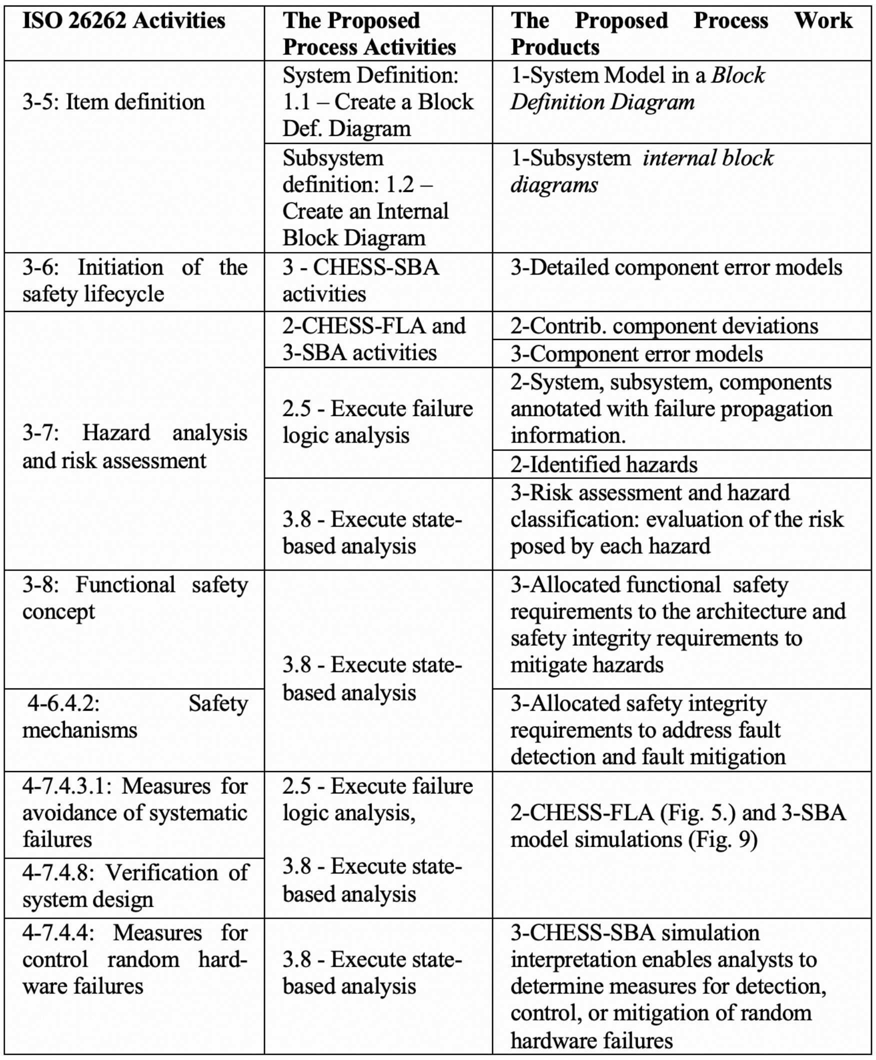
The systematic process described through this chapter was built upon activities and work products prescribed by ISO 26262 standard. In their work, Bressan et al. (2018) have compared the activities that are part from the proposed process with some of ISO 26262 Safety life-cycle activities.

The ”Architectural/System Model Design” phase described in Section 3.1, covers the ”3-5 Item Definition” activity prescribed by ISO 26262 as shown on Figure 3.12 by producing both a high-level and low-level definition of the system or product through SysML Block Definition and Internal Block Diagrams respectively. The ISO 26262 ”3-6 Initiation of the safety life-cycle” is covered by ”System Error Modeling” phase of the proposed process, since during this phase, components are annotated with dependability or failure logic information that will be further used to perform CHESS Failure Logic and State-Based Dependability Analyses. This support can be useful for reusing artifacts in different projects or environments since a set of different behaviors that can be raised when the artifact’s environment changes (BRESSAN et al., 2018) can be stored in different error models.

Both CHESS-SBA and CHESS-FLA support engineers with useful information about potential threats that may impact the overall system safety, through the results of their analyses. Both the "Failure Logic Analysis" and "State-Based Dependability Analysis" phases described in Section 3.2 of this report, cover ISO 26262's "3-7 Hazard Analysis and Risk Assessment" activity since they aid, through component error models and analyses, with the identification of potential hazardous behaviours that can impact on the overall system safety, and how these behaviours propagate throughout system architecture. CHESS-SBA's State-Based Dependability Analysis results can also be used to calculate the level of exposure to each identified hazard. Thus, supporting engineers during risk classification, covering ISO 26262 "3-7 Hazard Analysis and Risk Assessment" activity (BRESSAN et al., 2018).

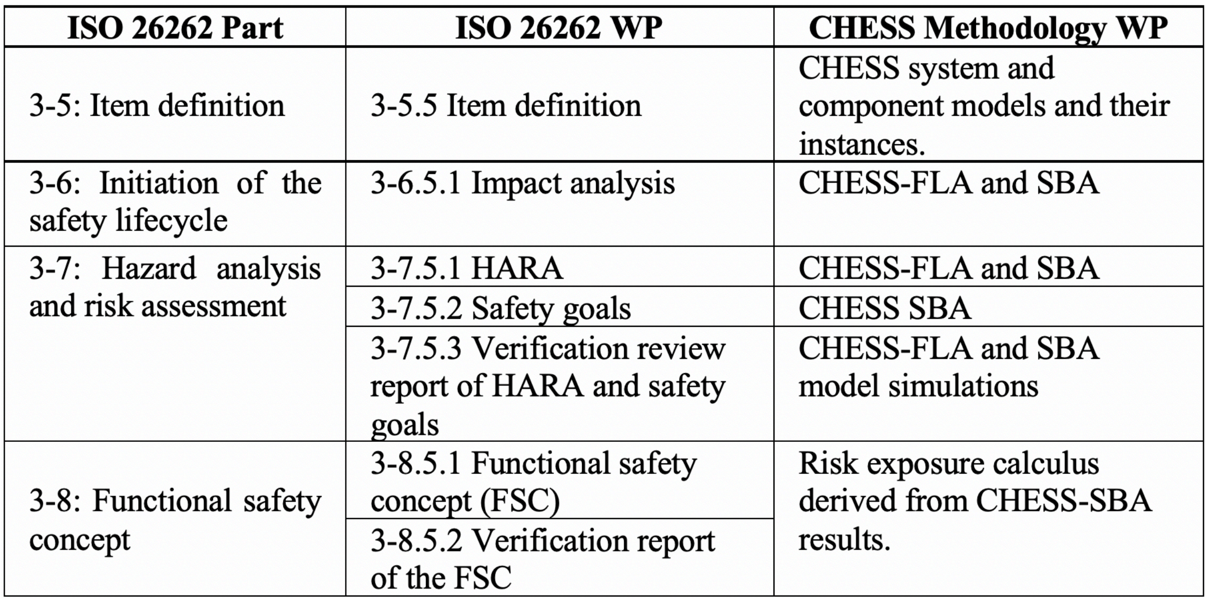
The State-Based Dependability Analysis and Failure Logic Analysis phases can also address ISO 26262's "3-8 Functional Safety Concept" and "4-6.4.2 Safety Mechanisms" activities. System Engineers and Analysts can use CHESS-FLA and CHESS-SBA model simulation results to allocate functional safety requirements and Automotive Safety Integrity Levels to mitigate the identified hazards through the system architecture. The results generated by these two analyses can be also used to determine the impact of item failures in the overall safety of the system. These results can be also used to aid them on determining ways to control random item failures thus, addressing ISO 26262's "4-7.4.3.1 Measures for avoidance of systematic failures" and "4-7.4.4 Measures to control random hardware failures" activities (BRESSAN et al., 2018).

At last, the application of both State-Based Dependability Analysis and Failure Logic Analysis phases and their prescribed activities address the "4-7.4.8 Verification of system design" activity defined in the ISO 26262 standard since through these analyses results, analysts can verify if the system requirements are all covered by its developed architecture (BRESSAN et al., 2018). The relation between the approach described in this chapter and ISO 26262 activities is shown in Figure 3.12 and their work products in Figure 3.13:



ISO 26262 Activities	**The Proposed Process Activities**	**The Proposed Process Work Products**
3-5: Item definition	System Definition: 1.1 – Create a Block Def. Diagram	1-System Model in a *Block Definition Diagram*
	Subsystem definition: 1.2 – Create an Internal Block Diagram	1-Subsystem *internal block diagrams*
3-6: Initiation of the safety lifecycle	3 - CHESS-SBA activities	3-Detailed component error models
3-7: Hazard analysis and risk assessment	2-CHESS-FLA and 3-SBA activities	2-Contrib. component deviations
		3-Component error models
	2.5 - Execute failure logic analysis	2-System, subsystem, components annotated with failure propagation information.
		2-Identified hazards
	3.8 - Execute state-based analysis	3-Risk assessment and hazard classification: evaluation of the risk posed by each hazard
3-8: Functional safety concept	3.8 - Execute state-based analysis	3-Allocated functional safety requirements to the architecture and safety integrity requirements to mitigate hazards
4-6.4.2: Safety mechanisms		3-Allocated safety integrity requirements to address fault detection and fault mitigation
4-7.4.3.1: Measures for avoidance of systematic failures	2.5 - Execute failure logic analysis,	2-CHESS-FLA (Fig. 5.) and 3-SBA model simulations (Fig. 9)
4-7.4.8: Verification of system design	3.8 - Execute state-based analysis	
4-7.4.4: Measures for control random hard-ware failures	3.8 - Execute state-based analysis	3-CHESS-SBA simulation interpretation enables analysts to determine measures for detection, control, or mitigation of random hardware failures

Figure 3.12: The described process and the ISO 26262 activities (BRESSAN et al., 2018)



ISO 26262 Part	ISO 26262 WP	CHESS Methodology WP
3-5: Item definition	3-5.5 Item definition	CHESS system and component models and their instances.
3-6: Initiation of the safety lifecycle	3-6.5.1 Impact analysis	CHESS-FLA and SBA
3-7: Hazard analysis and risk assessment	3-7.5.1 HARA	CHESS-FLA and SBA
	3-7.5.2 Safety goals	CHESS SBA
	3-7.5.3 Verification review report of HARA and safety goals	CHESS-FLA and SBA model simulations
3-8: Functional safety concept	3-8.5.1 Functional safety concept (FSC)	Risk exposure calculus derived from CHESS-SBA results.
	3-8.5.2 Verification report of the FSC	

Figure 3.13: CHESS and ISO 26262 Work Products (BRESSAN et al., 2018)

4 Reuse and Variability in CHESS

In this chapter, a preliminary approach to support variability in both CHESS architectural and safety artifacts, is presented. The proposed approach was formalized in a SPEM 2.0 activity diagram. Each one of the approach‘s phases is described in the following sections. The proposed process was validated through a case study using a slightly modified version of the HBS model presented in Chapter 3.

As shown on Figure 4.1, the proposed process comprises three main phases: Variability Specification, Variability Realization, and Variability Resolution. These phases are decomposed into 14 tasks, 4 activities, and generate 3 work products. The proposed approach demands the following inputs: a base model specified using the steps described in the System Design/Modeling phase (see Chapter 3) (BRESSAN et al., 2018). The starting point of the approach is the creation of a BVR Model. Then, there is the phase where a VSpec tree or diagram containing all the product's possible features is specified, realization which is when artifacts that will be kept in or removed from specific VSpecs are chosen and resolution, the phase where product variants and their features or VSpecs are defined and the derived model is generated based on those selections.

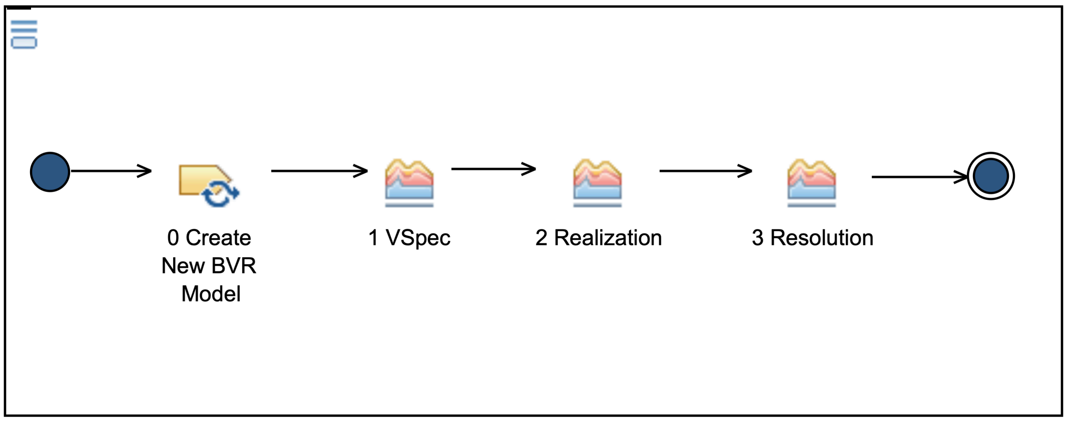


Figure 4.1: The proposed CHESS + BVR approach and their main phases

4.1 Architectural Model Variability Management

Inputs: CHESS project containing a Papyrus model, a BVR model and a system's architectural and error (optional) models graphically specified through Block Definition and Internal Block diagrams.

Outputs: Derived Papyrus model containing the desired product variant.

In order to derive a base model within a CHESS project using the BVR tool bundle, the model should be first specified by modeling its components, subcomponents, ports and connections using both SysML's Block Definition and Internal Block Definition diagrams. The base model must contain all the possible model elements that are part of some specific product configuration. These elements are subsequently kept in or derived out of the model depending on the variant that is being generated by the tool. Figure 4.2 shows the Block Definition diagram containing part of HBS's base model. This specific model contains both the *MechanicalPedal* components, one manufactured by Bosch and the other by Mercedes-Benz, that can be kept in a final derived model depending on the product variant in question. Then, a new BVR Model within the previously specified CHESS project must be created.

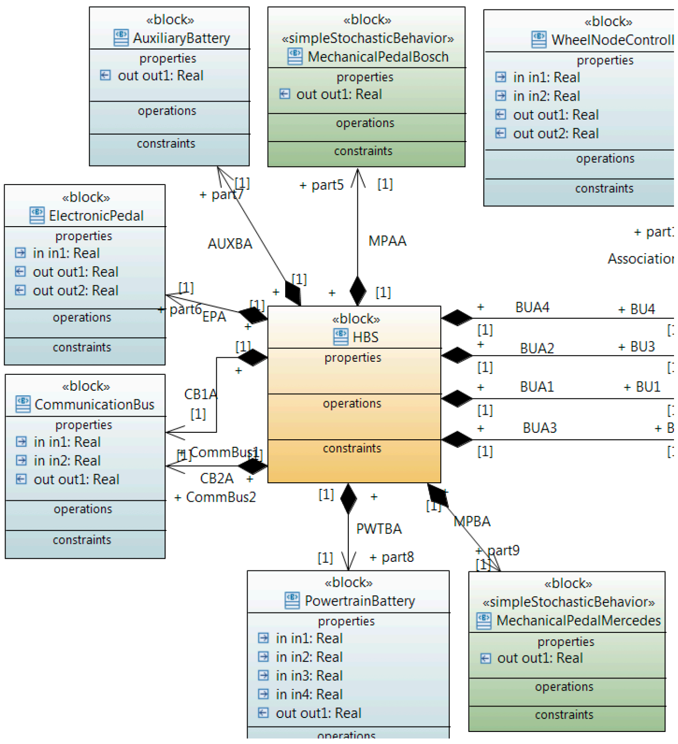

«block» AuxiliaryBattery
«simpleStochasticBehavior» MechanicalPedalBosch
«block» WheelNodeControll
«block» ElectronicPedal
«block» HBS
«block» CommunicationBus
«block» PowertrainBattery
«simpleStochasticBehavior» MechanicalPedalMercedes
AUXBA
MPAA
EPA
CB1A
CB2A
PWTBA
MPBA
BUA1
BUA2
BUA3
BUA4

Figure 4.2: Part of the Hybrid Braking System base model in a Block Definition Diagram

4.1.1 VSpec

Inputs: A BVR Model.

Outputs: The product's VSpec tree.

The VSpec phase is consisted of a task, an activity and a work product that is the final VSpec tree containing all the product's features or VSpecs as show on Figure 4.3:

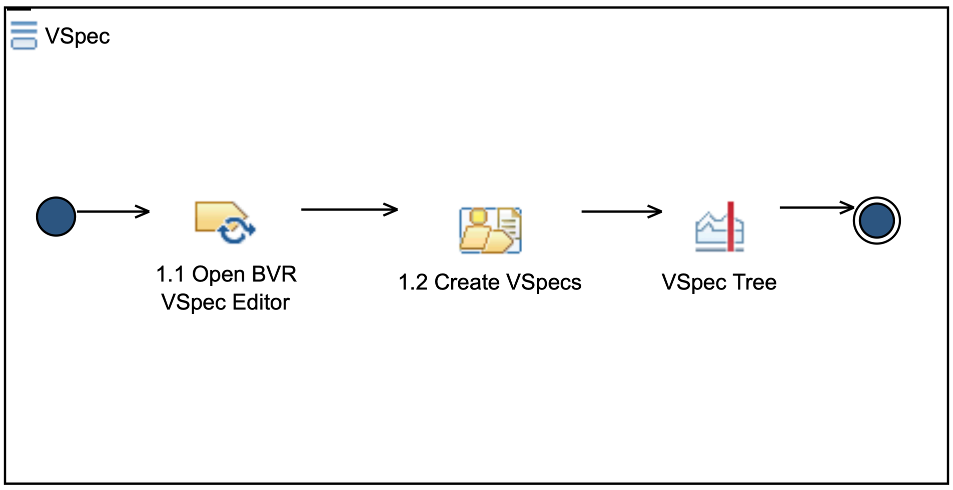


Figure 4.3: The process for specifying a VSpec tree in BVR

Once the BVR Model is created, the BVR VSpec Editor must be used to specify each different VSpec or each one of the possible configurations that the previously modeled base system can have using a VSpec tree. Figure 4.4 shows the VSpec tree for the HBS model used in this chapter to exemplify the process. The Hybrid Braking System in question has a total of 4 different possible configurations: one which it only contains the front Brake Units, one containing only the rear Brake Units, one that consists of two Brake Units which are set up diagonally and at last, a configuration containing all four Brake Units. Each configuration can also be consisted of any one of the 4 different types of *PowertrainBatteries* which only differ from each other in the number of input ports and in their error model. Different settings can also be composed of any one of the 2 different Mechanical Pedal models where each one have their own different manufacturer and «SimpleStochasticBehavior» parameter values. Each different configuration can imply in a different set of system components. The "*RearWheelsOnly*" variant for example, implies that the Powertrain Battery model to be used with this specific setting must be the B model and that the Mechanical Pedal must be the one manufactured by Bosch. For the "*FrontWheelsOnly*" variant, either one of the Mercedes-Benz or Bosch *MechanicalPedal* components can be used but the *PowertrainBattery* variant must be the "*PowerTranBatteryA*":

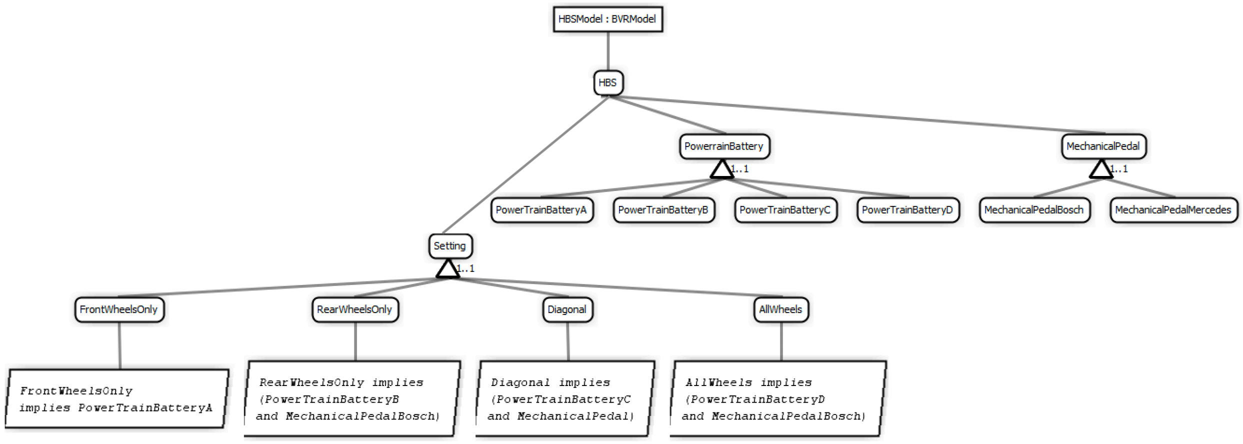


Figure 4.4: The HBS VSpec tree

4.1.2 Realization

Inputs: CHESS project containing a Papyrus model and a system's architecture.

Outputs: Product's variants, placements, replacements and links between the VSpecs and variants.

The next phase, the Realization phase, consists on using BVR's Realization Editor to specify which model elements will be kept in and which elements will be derived out in each one of the VSpecs specified in the previously created VSpec tree. As shown on Figure 4.5, this phase is consisted of 9 tasks, 2 activities and a work product which is the final specified Realization Model:

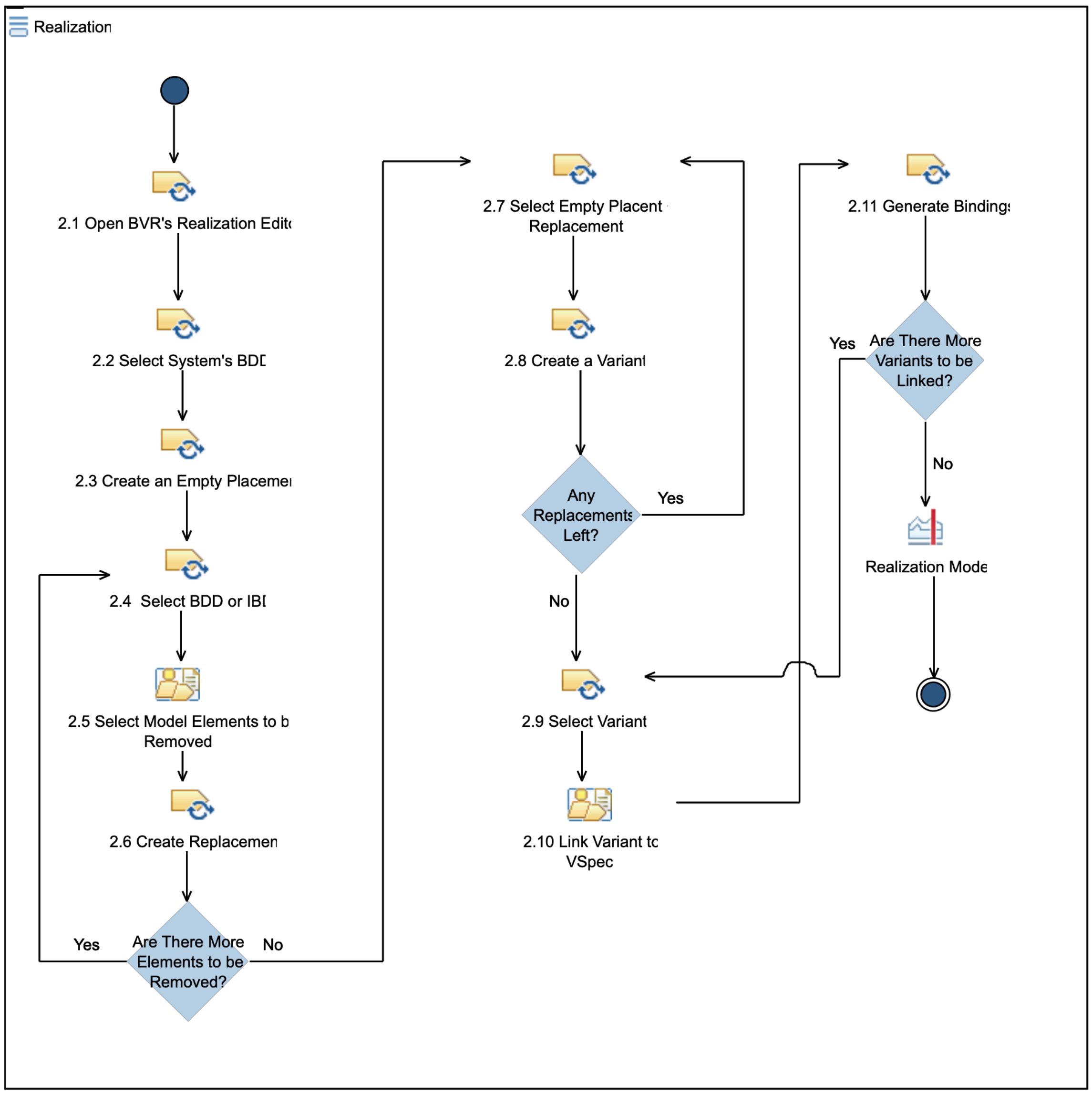
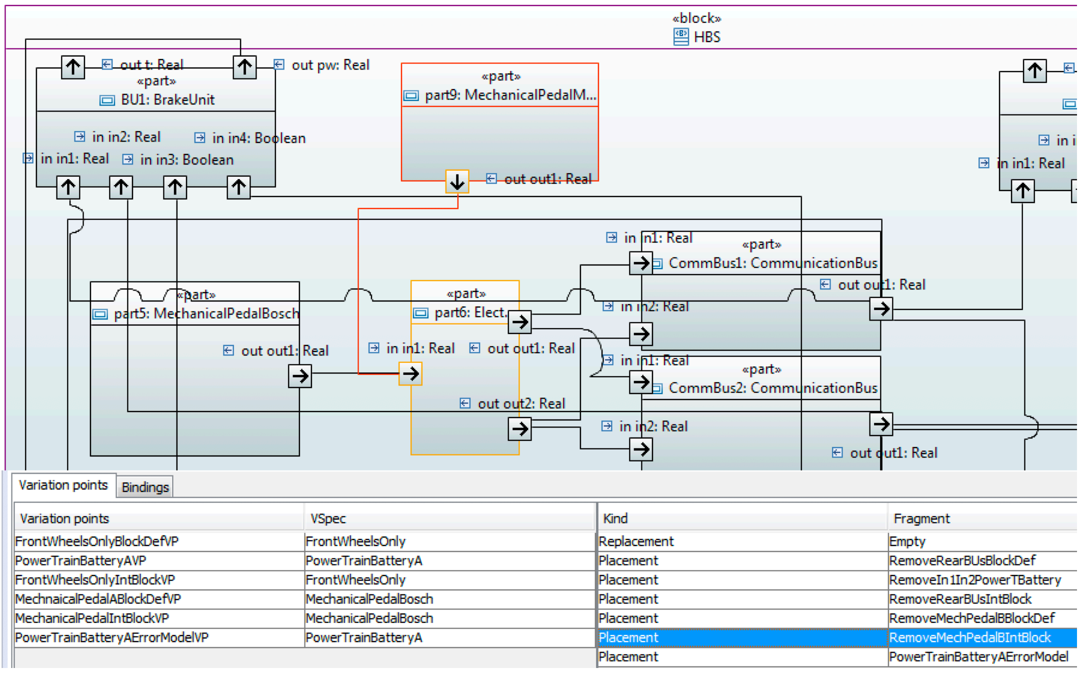

Realization
2.1 Open BVR's Realization Edit
2.2 Select System's BDD
2.3 Create an Empty Placeme
2.4 Select BDD or IBI
2.5 Select Model Elements to b Removed
2.6 Create Replacemen
Yes
Are There More Elements to be Removed?
No
2.7 Select Empty Placent Replacement
2.8 Create a Variant
Any Replacements Left?
Yes
No
2.9 Select Variant
2.10 Link Variant to VSpec
2.11 Generate Bindings
Yes
Are There More Variants to be Linked?
No
Realization Mode

Figure 4.5: The steps required to perform the Realization phase on a CHESS model using BVR

The first few tasks described in this phase, consist on selecting the system's Block Definition Diagram and creating an empty replacement. Now that an empty replacement has been created, elements that will be removed from the base model when an specific VSpec is selected, must be chosen. For the "*FrontWheelsBrakingOnly*" variant, placements containing both rear *BrakeUnits* must be created. The first placement "*RemoveRearBUsBlockDef*", is consisted of the two composition associations represented in the Block Definition diagram between the *HBS* and *BrakeUnit* «block»s. These associations represent the instances of the two rear *BrakeUnit*s inside the HBS «block». The second placement contains both the rear *BrakeUnit* «part»s or the *BrakeUnit*s instances represented in HBS' Internal Block diagram and all the connectors coming from or going to these two «part»s. The same steps were applied on the *PowertrainBattery* and *MechanicalPedal* components since this specific VSpec implies on using the *MechanicalPedal* made by Bosch and the *PowertrainBattery* model A containing only the input ports *in3* and *in4*.

Once all the placements are specified, placements and a replacements or in this case, placements and the previously created empty replacement should be combined together into a Variant and linked to a previously defined VSpec. A single VSpec can contain one or more variants linked to it. By doing so, BVR makes sure that when a certain VSpec is selected in the Resolution phase, all the model elements contained in a placement that linked to that VSpec, will be derived out of the final generated model. Figure 4.6 shows HBS's empty replacement, some of its placements, variants and to which VSpecs those variants are linked to. The image also shows a selected placement and the elements that will be derived out of the HBS' Internal Block diagram, highlighted in red, in case a VSpec containing a variation point that contains the "*RemoveMechPedalBIntBlock*" placement is selected.



Variation points	VSpec
FrontWheelsOnlyBlockDefVP	FrontWheelsOnly
PowerTrainBatteryAVP	PowerTrainBatteryA
FrontWheelsOnlyIntBlockVP	FrontWheelsOnly
MechnaicalPedalABlockDefVP	MechanicalPedalBosch
MechanicalPedalIntBlockVP	MechanicalPedalBosch
PowerTrainBatteryAErrorModelVP	PowerTrainBatteryA

Kind	Fragment
Replacement	Empty
Placement	RemoveRearBUsBlockDef
Placement	RemoveIn1In2PowerTBattery
Placement	RemoveRearBUsIntBlock
Placement	RemoveMechPedalBBlockDef
Placement	RemoveMechPedalBIntBlock
Placement	PowerTrainBatteryAErrorModel

Figure 4.6: Part of HBS' Realization model

4.1.3 Resolution

Inputs: Product variants and their VSpecs.

Outputs: Derived model containing the desired product variant.

The last phase of the proposed process involves creating product variants and selecting the VSpecs that will compose each one of them. The Resolution phase is composed by 3 tasks, one activity and one work product as shown on Figure 4.7:

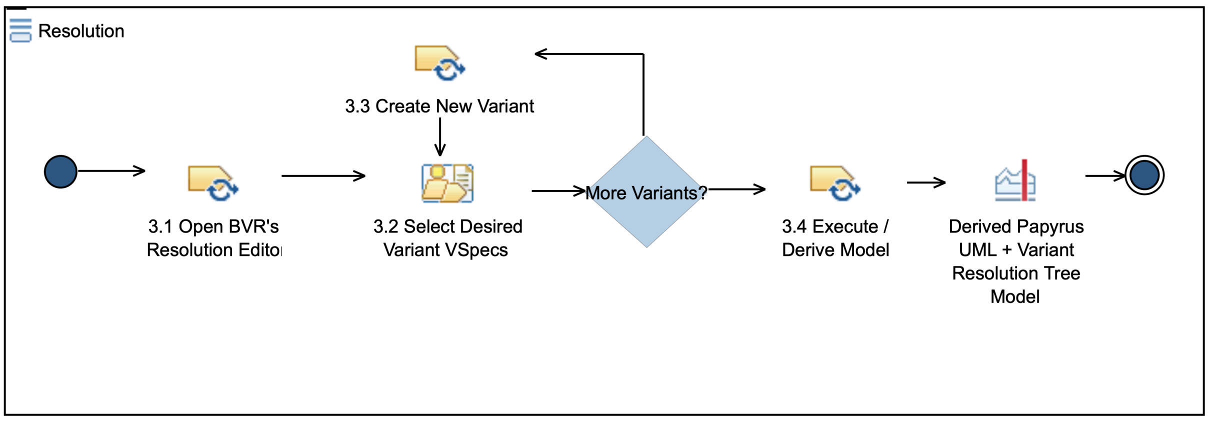

Resolution
3.3 Create New Variant
3.1 Open BVR's Resolution Editor
3.2 Select Desired Variant VSpecs
More Variants?
3.4 Execute / Derive Model
Derived Papyrus UML + Variant Resolution Tree Model

Figure 4.7: The steps required to perform the Resolution phase on BVR

The BVR Resolution editor displays a different version of the previously created VSpec tree. The Resolution editor allows users to set the VSpecs that will be kept on a variant's final derived model and the VSpecs that will be derived out of that variant's model by setting them as either 'true' or 'false'. Figure 4.8 shows one of HBS's product variants (HBS[0]). In this specific product variant, the "*FrontWheelsOnly*" VSpec is set to 'true' meaning that all the model elements selected as placements that are linked to this VSpec, will be removed from the base model when the final derived model is generated. Since this specific VSpec implies in "*PowertrainBatteryA*" and "*MechanicalPedalBosch*", both these VSpecs were also set as true:

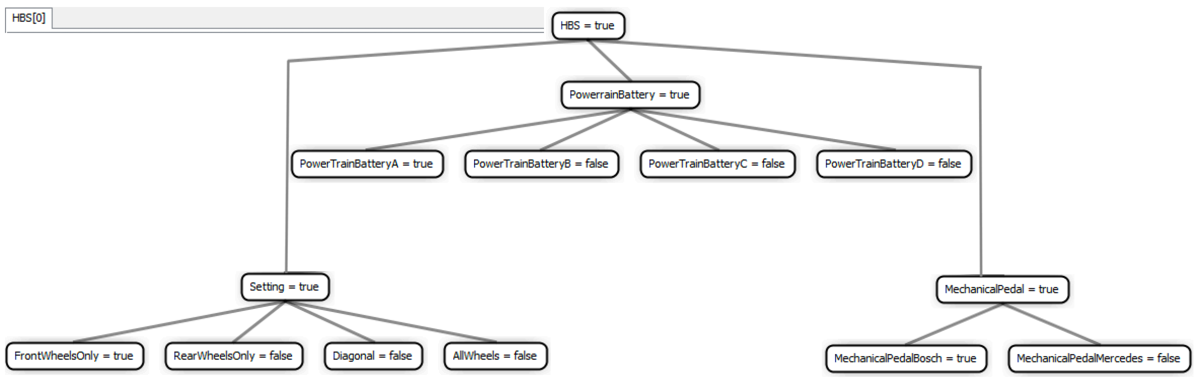

HBS[0]
HBS = true
PowerrainBattery = true
PowerTrainBatteryA = true
PowerTrainBatteryB = false
PowerTrainBatteryC = false
PowerTrainBatteryD = false
Setting = true
FrontWheelsOnly = true
RearWheelsOnly = false
Diagonal = false
AllWheels = false
MechanicalPedal = true
MechanicalPedalBosch = true
MechanicalPedalMercedes = false

Figure 4.8: HBS' Resolution model for the product variant HBS[0]

Once all the desired VSpecs have been selected, the Resolution model can be executed. Once done executing, the derived papyrus model containing the selected product variant is generated into the same CHESS project the base model is part of.

4.2 Component Error Model Variability Management

As previously mentioned in Chapter 3, there are three ways to add dependability and error propagation information to components in CHESS using certain CHESS-ML stereotypes. Components can behave differently depending on which product variant they are a part of and variability will be managed differently for each kind of error stereotype a component has. This section describes ways to manage variability in error models defined using all the different error stereotypes supported by the CHESS Toolset and uses the same HBS model from the previous sections to illustrate it.

The reason why it is important to define how each one of these stereotypes should be treated when it comes to variability management, is the fact that different product variants can imply on different error models or parameter values. A certain component, even though having the same architectural model across all product variants, can behave differently when integrated into a certain variant when compared to the others.

4.2.1 Components Stereotyped as «simpleStochasticBehavior» and «FLABehavior»

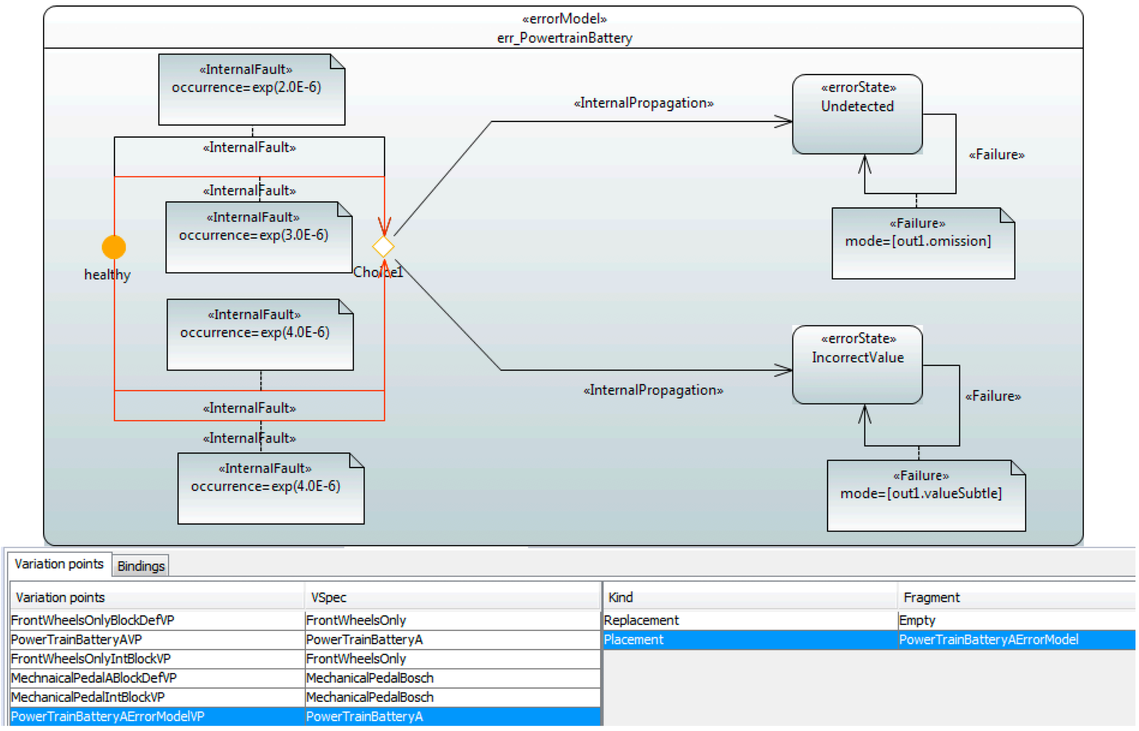
Variability management in components stereotyped as «simpleStochasticBehavior» and «FLABehavior», can be done by replicating the desired component, its ports and connectors according to the number of desired different error model variants. In the HBS model used in the previous sections for example, both *MechanicalPedal* components are «simpleStochasticBehavior» stereotyped. Even though both components share the same exact architectures (both components have one output port that outputs a Real value), they have different "*failureOccurrence*" values indicated by their corresponding manufacturers: the *MechanicalPedalBosch* has a "*failureOccurrence*" of exp(1.5E-6) and the *MechanicalPedalMercedes* has a "*failureOccurrence*" of exp(1.0E-6).

The process for managing variability in these components' error models follows the same tasks and activities described in the previous sections for architectural variability management. A base model should be defined containing all the replicated components and their corresponding error models and failure information. These components should be then selected as Placements or Replacements according to the VSpecs and product variants in question and finally, the base model should be derived according to the VSpecs selected in the Resolution phase.

4.2.2 Components Stereotyped as «ErrorModelBehavior»

There are a total of two ways to manage variability in components stereotyped as «ErrorModelBehavior». These components can be managed the same way as components stereotyped as «simpleStochasticBehavior» and «FLABehavior» by replicating the desired component, creating different error model state machines for each one of them and then selecting the component that will stay and the component that will be removed from the final derived model.

A better way to manage variability in «ErrorModelBehavior» stereotyped components though is by taking into account the fact that this specific stereotype uses a State Machine diagram to specify components' failure behaviors. By doing so, the same principle used to manage variability within components in the Internal and Block Definition Diagrams can be applied to on components' «ErrorModelBehavior» state machines by selecting transitions and states on the desired State Machine diagram and creating Placements and Replacements for the elements that must be kept in and the ones that must be removed from the base model. The Figure 4.9 shows a Variation Point linked to the VSpec "*PowertrainBatterA*". This specific VP, consists of an empty Replacement and a Placement that removes all the «InternalFault» transitions except the one with occurrence=exp(2.0E-6) from the base model. The base model in question, is consisted of 4 different «InternalFault» transitions, one for each different "*PowertrainBattery*" VSpec:



Variation points	VSpec
FrontWheelsOnlyBlockDefVP	FrontWheelsOnly
PowerTrainBatteryAVP	PowerTrainBatteryA
FrontWheelsOnlyIntBlockVP	FrontWheelsOnly
MechnaicalPedalABlockDefVP	MechanicalPedalBosch
MechanicalPedalIntBlockVP	MechanicalPedalBosch
PowerTrainBatteryAErrorModelVP	PowerTrainBatteryA

Kind	Fragment
Replacement	Empty
Placement	PowerTrainBatteryAErrorModel

Figure 4.9: PowertrainBattery base «errorModel» diagram and part of HBS' Realization model

5 Related Work

As a consequence of the increasing demand and the popularization of the development of products as part of software product lines by the industry, there are many different approaches in the literature trying to integrate variability management and Software Product Line Engineering techniques with tools or languages that support the model-driven development or even adapt or create a methodology to support variability management in models defined using a specific tool or language.

Among the many notable works in the area, there is the process proposed by (SHIRAISHI, 2010) where an approach for managing variability on models defined using the AADL language is defined. In this work, the author introduces a series of steps to support the variability management in systems defined through the mentioned language in the various different levels supported by AADL such as systems, processes and threads. The author proposes a way of managing variability in AADL models more specifically, in systems belonging to the automotive domain, using some reuse and inheritance concepts where components can have on or more distinct implementations where different variants of the same system can have different implementations of a given component depending of the variant in question.

In their work, Oliveira et al. (2018) introduce a model-based approach that enables the systematic reuse, in application engineering, of Software Product Line architecture and dependability artifacts called DEPendable-SPLE. This approach supports variability management in dependability analysis based on widespread Software Product Line Engineering methods. The main goal of this paper, is not only to present DEPendable-SPLE but also, to validate the presented approach through a case study using a critical system from the aerospace domain and prove that it makes 'efficient management of the impact of design and context variations on HARA and component fault modeling' (OLIVEIRA et al., 2018) possible.

Another notable work in the area is the OpenCert Toolset (OPENCERT, 2018) from AMASS project (AMASS, 2018). OpenCert is a solution that focuses on assurance certification management of Cyber-Physical Systems (CPS) (OPENCERT, 2018) and bundles not only an updated version of the CHESS toolset, but also the BVR tool bundle. In their work Javed e Gallina (2018), present how a seameless integration between the Eclipse Process Framework (EPF) and the BVR Tool Bundle can be achieved. The EPF Composer is a tool implemented using the Eclipse Process Framework and supports the specification of software processes. These processes are defined through elements such as activities, tasks and work products using the SPEM 2.0 notation. The authors also intend to extend the proposed integration to other tools such as CHESS and OpenCert for system and assurance case variability management respectively.

6 Conclusion

This chapter contains a summary of the contributions in this capstone project, its benefits and limitations, and future work that may concern and extend some aspects of the integration between the CHESS Toolset and the BVR tool bundle and variability management using other model-driven design tools.

6.1 Contributions

This work has presented a more elaborate version of the process introduced by Bressan et al. (2018) in their work. It has also focused on introducing how it is possible to use instances of «blocks»s as multiple «parts»s for components that are used more than once in a system's architecture. A more detailed Failure Logic Analysis case study containing two distinct scenarios and a more detailed description of each component's failure propagation behavior has been used to better illustrate the FLA process.

Additionally, this work has also presented an approach to support variability management in CHESS models. The integration between the CHESS Toolset and the BVR Tool Bundle has been validated by successfully applying the proposed process in a realistic case study. Different ways to manage variability in CHESS component error models have also been presented and exemplified using the Hybrid Braking System.

6.2 Benefits

Both systematic processes described in this work and the case studies offer a more practical and "hands on" way to understand how systems can be specified how certain analysis can be performed upon those systems and how their results can be interpreted using the CHESS toolset. The processes also guide users on managing variability in CHESS models using the BVR Tool Bundle on both architectural and error models in a completely cost free way since both tools are not commercial.

6.3 Limitations

Although the proposed process was proved to work, there may be some means to simplify the way variability management on CHESS error models is done by making some changes on how error information is attached to components. This could be done by letting the user attach more than one of the same or different error stereotypes to a single component eg. using SysML comments with the desired error stereotype. This would prevent the need to replicate components and their architectures in order to specify their different «simpleStochasticBehavior» and «FLABehavior» parameters in the base model.

6.4 Future Work

Additional case studies will be further developed to validate the approaches presented in this paper and a more detailed analysis of the proposed process against automotive and aerospace safety standards will be further performed.

Bibliography

AKTINSON, C.; KUHNE, T. Model-driven development: a metamodeling foundation. In: . [S.l.: s.n.], 2003. v. 20, p. 36–41.

ALI, R. “safety life cycle” – implementation benefits and impact on field devices. In: . FIELDVUE Business Development, Emerson Process Management - Fisher Controls Int’l., LLC. Austin, TX 78717: Department of Computer Science, 2005.

AMASS. Architecture-driven, multi-concern and seamless assurance and certification of cyber-physical systems. In: . [s.n.], 2018. Accessed: 2018-11-07. Disponível em: <https://www.amass-ecsel.eu>.

BACHMANN, F. et al. A meta-model for representing variability in product family development. In: *”Software Product-Family Engineering”*. ”Berlin, Heidelberg”: ”Springer Berlin Heidelberg”, 2004. p. ”66–80”.

BATTEUX, M. et al. The altarica 3.0 project for model-based safety assessment. In: *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*. [S.l.: s.n.], 2013. p. 741–746.

BIGLEVER. Software product lines matures into the next generation of systems and software product line engineering. In: . [s.n.], 2013. Accessed: 2018-04-20. Disponível em: <http://www.softwareproductlines.com>.

BRESSAN, L. P. et al. A systematic process for applying the chess methodology in the creation of certifiable evidence. In: *14th European Dependable Computing Conference*. [S.l.: s.n.], 2018.

CASTRO, R. de; ARAUJO, R. E.; FREITAS, D. Hybrid abs with electric motor and friction brakes. In: *22nd International Symposium on Dynamics of Vehicles on Roads and Tracks, (IAVDS11)*. [S.l.: s.n.], 2011.

CCOHS. Risk assessment. In: . Canadian Centre of Occupational Health and Safety, 2018. Accessed: 2018-09-21. Disponível em: <https://www.ccohs.ca/oshanswers/hsprograms/risk_assessment.html>.

CZERNY, B. J. et al. Iso 26262 functional safety draft international standard for road vehicles: Background, status, and overview. In: . [S.l.: s.n.], 2010.

FLEUREY, F. Langage et m´ethode pour une ing´enierie des mod‘eles fiabe”. In: . [S.l.: s.n.], 2006.

FRAKES, W.; ISODA, S. Success factors of systematic reuse. In: . [S.l.]: IEEE Software, 1994. p. 14–19.

FREEMAN, P. Reusable software engineering: Concepts and research directions. In: . [S.l.: s.n.], 1983.

GABRY, O. E. Software engineering—software process and software process models (part 2). In: . [s.n.], 2017. Accessed: 2018-04-20. Disponível em: <https://medium.com/omarelgabrys-blog/software-engineering-software-process-and-software-process-models-part-2-4a9d06213fdc>.

GALLINA, B.; PUNNEKKAT, S. Fi4fa: A formalism for incompletion, inconsistency, interference and impermanence failures' analysis. In: *Proceedings - 37th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2011*. [S.l.: s.n.], 2011. p. 493 – 500.

HABLI, I.; KELLY, T. P.; PAIGE, R. Functional hazard assessment in product-lines – a model-based approach. In: . [S.l.: s.n.], 2009.

HARPERCOLLINS. Collins english dictionary. In: . [S.l.]: HarperCollins Publisher, 2018.

HAUGEN Øystein; ØGåRD, O. Bvr – better variability results. In: . SINTEF, P.O. Box 124 Blindern, NO-0314 Oslo, Norway: Springer International Publishing Switzerland, 2014.

IEC, I. E.-T. C. Bs iec 61508 – functional safety of electrical/electronic/programmable electronic safety-related system. In: . [S.l.]: British Standards Institute/IEC, 2010.

IEC, I. E.-T. C. Electropedia: The world's online electrotechnical vocabulary. In: . [s.n.], 2015. Accessed: 2018-11-15. Disponível em: <http://www.electropedia.org/iev/iev.nsf/index?openform&part=192>.

INTECS. Concerto toolset user guide. In: . [S.l.: s.n.], 2016. p. 11–12.

ISO. Iso 26262: Road vehicles functional safety. In: . [S.l.]: ISO, 2011.

JAVED, M. A.; GALLINA, B. Safety-oriented process line engineering via seamless integration between epf composer and bvr tool. In: *Proceedings of the 22Nd International Systems and Software Product Line Conference - Volume 2*. New York, NY, USA: ACM, 2018. (SPLC '18), p. 23–28. ISBN 978-1-4503-5945-0.

KLEINER, M.; ALBERT, P.; BÉZIVIN, J. Parsing sbvr-based controlled languages. In: *Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009*. [S.l.: s.n.], 2009. p. 122 – 136.

KRUEGER, C. W. Software reuse. In: . [S.l.: s.n.], 1992.

LEVESON, N. G. Software safety: why, what, and how. In: . [S.l.]: ACM Surveys, 1986.

LEVESON, N. G. Safeware system safety and computers. In: . Addison-Wesley, Reading, MA: [s.n.], 1995.

LITTLEWOOD, B.; STRIGINI, L. Validation of ultrahigh dependability for software-based systems. In: . New York, NY, USA: ACM, 1993. v. 36, n. 11, p. 69–80.

MOD. Def-stan 00-56 issue 4 part 1: Safety management requirements for defense systems. In: . [S.l.]: Technical report, UK Ministry of Defense, 2007.

MTU. Hazard analysis: Using the hazard identification checklist. In: . Michigan Tech University, 2018. Accessed: 2018-09-21. Disponível em: <https://www.mtu.edu/ehs/docs/hazard-analysis-instructions.pdf>.

NASA. Fault tree analysis with aerospace applications. In: *Technical report, an update to NUREG-0492*. Washington, DC, USA: NASA Office of Safety and Mission Assurance, 2002.

NASTOV, B. Grammar and graphical concrete syntaxes generator assistant for domain specific modeling languages. In: . [S.l.: s.n.], 2013.

NRC. Fault tree handbook. In: . [S.l.]: Office of Nuclear Regulatory Research, 1981.

OLIVEIRA, A. L. A model-based approach to support the systematic reuse and generation of safety artefacts in safety-critical software product line engineering. In: . Instituto de Ciências Matemáticas e de Computação, USP, São Carlos, Brazil: [s.n.], 2016.

OLIVEIRA, A. L. de et al. Variability management in safety-critical software product line engineering. In: . [S.l.: s.n.], 2018.

OMG, O. M. G. Software systems process engineering meta-model specification (spem). In: . [S.l.: s.n.], 2008.

OPENCERT. Opencert. In: . [s.n.], 2018. Accessed: 2018-11-07. Disponível em: <https://www.polarsys.org/opencert/>.

PAPADOPOULOS, Y. et al. Engineering failure analysis and design optimization with hip-hops. In: *Journal of Engineering Failure Analysis*. [S.l.: s.n.], 2011. v. 18, p. 590–608.

PAPADOUPOULOS, Y.; MCDERMID, J. A. The potential for a generic approach to certification of safety critical systems in the transportation sector. In: . [S.l.: s.n.], 1999. p. 47–66.

PHOL, K.; BöCKLE, G.; LINDEN, F. van der. Software product line engineering. In: . [S.l.]: Springer, 2005.

RTCA. Do-178c software considerations in airborne systems and equipment certification. In: . [S.l.]: Radio Technical Commission for Aeronautics, 2012.

SEFER, E. A model-based safety analysis approach for high-integrity socio-technical component-based systems. In: . [S.l.]: Mälardalen University, 2015.

SHIRAISHI, S. An aadl-based approach to variability modeling of automotive control systems. In: . Toyota InfoTechnology Center Co., Ltd. Akasaka 6-6-20, Minato-ku Tokyo, Japan 107-0052: [s.n.], 2010.

SINTEF. Bvr tool. In: . [s.n.], 2018. Accessed: 2018-04-20. Disponível em: <https://bvr-tool.sintef.cloud>.

SLJIVO, I. et al. A method to generate reusable safety case argument-fragments from compositional safety analysis. In: . [S.l.]: Journal of Systems Software, 2016.

SOMMERVILLE, I. Dependability. In: . [S.l.: s.n.], 2000.

SOMMERVILLE, I. Critical systems. In: . [s.n.], 2018. Accessed: 2018-04-20. Disponível em: <http://iansommerville.com/software-engineering-book/web/critical-systems/>.

SUN, L. Establishing confidence in safety assessment evidence. phd thesis. In: . The University of York, York, United Kingdom: Department of Computer Science, 2013.

UNIFI. State-based dependability analysis. In: . [S.l.]: Università di Firenze, 2017.

VERMA, A. K.; AJIT, S.; KARANKI, D. R. Reliability and safety engineering. In: . 2. ed. [S.l.]: Springer, 2015, (Springer Series in Reliability Engineering).

VILLELA, K. B. et al. A survey on software variability management approaches. In: . [S.l.: s.n.], 2014.