UNIVERSIDADE FEDERAL DE JUIZ DE FORA
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

**Lucas Paiva Bressan**

# CRITIVAR: A Variability Modeling and Transformation Approach for Safety and Security-Critical Systems

Juiz de Fora
2023

**Lucas Paiva Bressan**

# CRITIVAR: A Variability Modeling and Transformation Approach for Safety and Security-Critical Systems

Dissertação apresentada ao Programa de Pós Graduação em Ciência da Computação da Universidade Federal de Juiz de Fora como requisito parcial à obtenção do título de Mestre em Ciência da Computação. Área de concentração: Engenharia de Software e Banco de Dados

Orientador: Prof. Dr. André Luiz de Oliveira

Coorientador: Profa. Dra. Fernanda Claudia Alves Campos

Juiz de Fora

2023

Ficha catalográfica elaborada através do programa de geração
automática da Biblioteca Universitária da UFJF,
com os dados fornecidos pelo(a) autor(a)

Paiva Bressan, Lucas.
CRITIVAR: A Variability Modeling and Transformation Approach for Safety and Security-Critical Systems / Lucas Paiva Bressan. -- 2023.
113 f.

Orientador: Andre Luiz de Oliveira
Coorientadora: Fernanda Claudia Alves Campos
Dissertação (mestrado acadêmico) - Universidade Federal de Juiz de Fora, Instituto de Ciências Exatas. Programa de Pós-Graduação em Ciência da Computação, 2023.

1. Functional Safety. 2. Cybersecurity. 3. Dependability. 4. Product Lines. 5. Variability. I. de Oliveira, Andre Luiz, orient. II. Alves Campos, Fernanda Claudia, coorient. III. Título.

# UNIVERSIDADE FEDERAL DE JUIZ DE FORA

## PRÓ-REITORIA DE PÓS-GRADUAÇÃO E PESQUISA

## ATA DE DEFESA DE TRABALHO DE CONCLUSÃO DE PÓS-GRADUAÇÃO *STRICTO SENSU*

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

**N° PROPP:** 659.19102023.4-M

**N° PPG: 155**

**AVALIAÇÃO DA BANCA EXAMINADORA**

Tendo o senhor Presidente declarado aberta a sessão, mediante o prévio exame do referido trabalho por parte de cada membro da Banca, o discente procedeu à apresentação de seu Trabalho de Conclusão de Curso de Pós-graduação *Stricto sensu* e foi submetido à arguição pela Banca Examinadora que, em seguida, deliberou sobre o seguinte resultado:

**(X) APROVADO (Conceito A)**

( ) **APROVADO CONDICIONALMENTE (Conceito B)**, mediante o atendimento das alterações sugeridas pela Banca Examinadora, constantes do campo Observações desta Ata.

( ) **REPROVADO (Conceito C)**, conforme parecer circunstanciado, registrado no campo Observações desta Ata e/ou em documento anexo, elaborado pela Banca Examinadora

Novo título da Dissertação/Tese (só preencher no caso de mudança de título):

Observações da Banca Examinadora caso:

- O discente for Aprovado Condicionalmente
- Necessidade de anotações gerais sobre a dissertação/tese e sobre a defesa, as quais a banca julgue pertinentes.

Nada mais havendo a tratar, o senhor Presidente declarou encerrada a sessão de Defesa, sendo a presente Ata lavrada e assinada pelos senhores membros da Banca Examinadora e pelo discente, atestando ciência do que nela consta.

**INFORMAÇÕES**

- Para fazer jus ao título de mestre(a)/doutor(a), a versão final da dissertação/tese, considerada Aprovada, devidamente conferida pela Secretaria do Programa de Pós-graduação, deverá ser tramitada para a PROPP, em Processo de Homologação de Dissertação/Tese, dentro do prazo de 90 dias a partir da data da defesa. Após a entrega dos dois exemplares definitivos, o processo deverá receber homologação e, então, ser encaminhado à CDARA.

- Esta Ata de Defesa é um documento padronizado pela Pró-Reitoria de Pós-Graduação e Pesquisa. Observações excepcionais feitas pela Branca Examinadora poderão ser registradas no campo disponível acima ou em documento anexo, desde que assinadas pelo(a) Presidente(a).

- Esta Ata de Defesa somente poderá ser utilizada como comprovante de titulação se apresentada junto á Certidão da Coordenadoria de Assuntos e Registros Acadêmicos da UFJF (CDARA) atestando que o processo de confecção e registro do diploma está em andamento.

BANCA EXAMINADORA

**Prof. Dr. André Luiz de Oliveira** - Orientador
Universidade Federal de Juiz de Fora (UFJF)

**Profa. Dra. Fernanda Cláudia Alves Campos** - Coorientadora
Universidade Federal de Juiz de Fora (UFJF)

**Profa. Dra. Genaína Nunes Rodrigues** - Membro Externo
Universidade de Brasília (UnB)

**Profa. Dra. Rosana Teresinha Vaccare Braga** - Membro Externo
Universidade de São Paulo (USP)

**Profa. Dra. Regina Maria Maciel Braga Villela** - Membro Interno
Universidade Federal de Juiz de Fora (UFJF)

Juiz de Fora, 16 / 10 / 2023.

Documento assinado eletronicamente por **Andre Luiz de Oliveira**, **Professor(a)**, em 20/10/2023, às 14:24, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Rosana Teresinha Vaccare Braga**, **Usuário Externo**, em 20/10/2023, às 14:42, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Regina Maria Maciel Braga Villela**, **Professor(a)**, em 20/10/2023, às 14:58, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Fernanda Claudia Alves Campos**, **Professor(a)**, em 20/10/2023, às 16:54, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Genaína Nunes Rodrigues**, **Usuário Externo**, em 26/10/2023, às 10:16, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Lucas Paiva Bressan**, **Usuário Externo**, em 31/10/2023, às 15:07, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

A autenticidade deste documento pode ser conferida no Portal do SEI-Ufjf (www2.ufjf.br/SEI) através do ícone Conferência de Documentos, informando o código verificador **1526884** e o código CRC **018BF26E**.

Dedico este trabalho aos meus pais, esposa e todos os outros que me deram suporte durante minha trajetória.

# RESUMO

A customização em massa de sistemas críticos modernos tem gerado produtos com milhares de pontos de variação, impactando arquitetura and propriedades de dependability (e.g., safety e security) destes sistemas. Devido ao fato das técnicas existentes de gerencia de variabilidades serem rudimentares, modelar a diversidade de sistemas críticos e suas propriedades de dependability se torna uma tarefa árdua. Por isso, esta dissertação apresenta uma abordagem de gerência de variabilidades para sistemas críticos. A abordagem é ilustrada através de um estudo de caso de domínio automotivo. Como avaliação, a abordagem foi comparada com duas técnicas similares de gerencia de variabilidades. Dentre os benefícios, a abordagem apresentada reduz o "gap" entre variabilidade e anotações de dependability de granulosidade baixa e garante a derivação de modelos corretos e completos.

Palavras-Chave: Functional Safety. Cybersecurity. Dependability. Linhas de Produto. Variabilidade.

# ABSTRACT

Mass customization of modern critical systems has led to products with many variation points, impacting their architecture and dependability properties. Given the rudimentary vulnerability management techniques available, modeling diversity in critical systems and their corresponding dependability (e.g., safety and security) characteristics is challenging. That being said, this dissertation presents a novel annotative variability management approach for critical systems. The approach is illustrated using a case study from the automotive domain. As part of the evaluation, the proposed approach is compared against two similar vulnerability management solutions. Among the benefits, the the proposed approach reduces the gap between variability and finer-grained model dependability annotations and ensures the derivation of correct and complete critical system models.

Keywords: Functional Safety. Cybersecurity. Dependability. Product Lines. Variability.

List of Figures

List of Tables

# LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| SPL | Software Product Line |
| SPLE | Software Product Line Engineering |
| FLA | Failure Logic Analysis |
| SBA | State-Based Analysis |
| MOF | Meta-Object Facility |
| UML | Unified Modeling Language |

Contents

# 1 INTRODUCTION

## 1.1 CONTEXT

Safety-critical systems are systems whose failure or malfunction could result in loss of human life, significant damage to the property or damage to the environment (KNIGHT, 2002). There are many well known examples of safety-critical systems in application domains such as medical devices (ROBERTS et al., 2017), automotive (GREENGARD, 2015), aircraft flight control (STOREY, 1996), railway (AUSTON, 2021) (RETP, 2021), defense (EMBRAER, 2021) and nuclear power plant (BYVAIKOV et al., 2006) control systems. Failure in one of these systems might endanger human life, leading to economic loss, legal retaliation and environmental damage.

The design of safety-critical systems have been moved from mechanical-based towards complex computer-reliant architectures. Computer-based system architectures can perform sophisticated control functions and their applications are immerse in many areas that affect our daily lives (KNIGHT, 2002). The vehicle we drive contains multiple programmable components[1] that perform safety-critical tasks such as engine control, battery management, and steering control. Computer-based systems boost advanced safety-critical features in automotive systems such as electronic stability control that automatically detects loss of steering control and brakes individual wheels to fix the vehicle's trajectory (AZEVEDO, 2015; GREENGARD, 2015).

Apart from the benefits of automation, computer-based systems introduce additional complexity into the system architecture. Complex systems are more difficult to design and more likely to contain errors (STOREY, 1996). For example, hardware architectures are composed by several small parts that can fail in multiple ways, rising a huge number of failure scenarios to be considered. With respect to software, the number of execution paths can easily increase into a number that is not feasible to test exhaustively (AZEVEDO, 2015). Currently, there is a trend towards increasing the complexity of safety-critical software (BOZZANO; VILLAFIORITA, 2010), especially in the automotive domain. Such complexity is inherent the greater possibilities offered by inter-connectivity and the increased computing power (MACGREGOR; BURTON, 2018). In the development of the latest automotive systems, newer functionalities are provided by integrating independent Original Equipment Manufacturers (OEMs).

The mass customization in the automotive industry (SPLC HALL OF THE FAME, 2019) (SCHULZE; MAUERSBERGER; BEUCHE, 2013; TISCHER et al., 2011) and other safety-critical domains (WÖLFL et al., 2015; DORDOWSKY; BRIDGES; TSCHOPE, 2011; HABLI; KELLY; HOPKINS, 2007) has led to a higher diversity within single

[1] Component: a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment (OMG, 2017c).

products with hundreds and thousands of variations points (POHL; HÖCHSMANN, et al., 2018). A variation point consists in a place in an artefact that can be changed or customized to be used in different contexts (GURP; BOSCH; SVAHNBERG, 2001). The difference between configurations of an artefact can be found in the presence or absence of a certain optional or alternative model[2]/code elements, parameters or parameter values (SCHULZE; MAUERSBERGER; BEUCHE, 2013). A variation point describes all possible configurations available at a given point in an artefact.

Automotive electronic control units (ECUs) (TISCHER et al., 2011) used in airbags, electronic window lifter and driver assistant systems, and powertrain controllers (SPLC HALL OF THE FAME, 2019) are highly variant-intensive. With the purpose to cope with the complexity of modern safety-critical software with respect to the increasingly number of variants, component-based (OMG, 2017a,c) approaches and Software Product Lines (SPL) (CLEMENTS; NORTHROP, 2001) have been successfully used in industry, specially in the automotive (SCHULZE; MAUERSBERGER; BEUCHE, 2013; TISCHER et al., 2011) and aerospace (WÖLFL et al., 2015; DORDOWSKY; BRIDGES; TSCHOPE, 2011) domains, increasing the software quality, reducing the time-to-market and system development costs.

Although product line design maximizes the reuse across multiple critical software variants, it must still yield safe individual configurations. A malfunction or loss of a function in a safety-critical product can potentially impact on functional safety with severe consequences to the system operating environment, threatening human's life with serious injuries or even loss of life.

The nature and inherent complexity of critical systems demands addressing dependability requirements, e.g., safety, cybersecurity, reliability, availability, integrity and maintainability (AVIZIENIS; LAPRIE; RANDELL, et al., 2004), to demonstrate the services provided by these systems can justifiably be trusted. Justifying why a system is acceptably safe and its components are reliable before release for operation is key to certification and the adoption of the system by the market, and those are questions to be answered within the scope of Dependability Engineering.

## 1.2 MOTIVATION

The development of complex and variant-intensive critical systems demands the analysis of safety, cybersecurity, reliability, and other dependability properties of system functions and components, and ensuring that they hold across different configurations in compliance with standards.

Variation in the system design and its intended usage context[3] may impact on

[2] A model is an abstraction with a well defined purpose.

[3] Usage Context: refers to characteristics of the operating environment that determine how

dependability properties, modifying potential events leading to accidents, their risks e.g., in terms of severity and likelihood, the dependability requirements to mitigate their effects, the ways how failures propagate throughout architectural subsystems and components contribute to system failures. So, reusing critical components across different configurations requires the systematic reuse of associated dependability artefacts. Dependability artefacts may contain qualitative information about component faults and their propagation's expressed in failure logic models (FLM) (GALLINA; JAVED, et al., 2012; PAPADOPOULOS et al., 2011; FEILER, 2013), and quantitative information expressed in probabilistic models (MONTECCHI; PURI, 2020; MALHOTRA; TRIVEDI, 1995).

The dependability information is the key point of diversity in the analysis and verification of complex and configurable critical system architectures (BRESSAN; OLIVEIRA; CAMPOS; CAPILLA, 2021; OLIVEIRA; BRAGA, et al., 2019). Reusing qualitative failure logic and probabilistic models for system dependability analysis demands a way to manage the diversity of emergent failure events leading to accidents, cybersecurity threats, safety and security goals and requirements that may rise in different configurations and targeting usage contexts (OLIVEIRA; BRAGA, et al., 2019; OLIVEIRA, André Luiz de; BRAGA, R.; MASIERO, P.; PAPADOPOULOS, et al., 2018).

Although safety and cybersecurity standards, e.g., ISO 26262 and ISO/SAE 21434 provide guidelines to develop single products, industrial production is inherently variable in which different critical products are built upon a common base system (WOLSCHKE et al., 2019; SPLC HALL OF THE FAME, 2019; WÖLFL et al., 2016; TISCHER et al., 2011). SPL approaches have been successfully adopted by automotive (SPLC HALL OF THE FAME, 2019; TISCHER et al., 2011) and aerospace (WÖLFL et al., 2016; DORDOWSKY; BRIDGES; TSCHOPE, 2011) industry to cope with the inherent complexity and variability of safety-critical systems.

Ensuring the proper use and re-use of product family components across variants demand the integration of functional safety and cybersecurity engineering within SPL processes (POHL; BÖCKLE; LINDEN, 2005; GOMAA, 2004; KANG, K. C. et al., 1998), and variant management on dependability artifacts (OLIVEIRA; BRAGA, et al., 2019; OLIVEIRA, André Luiz de; BRAGA, R.; MASIERO, P.; PAPADOPOULOS, et al., 2018). However, the manual analysis of SPL dependability properties and demonstration that they hold across all possible variants and contexts would be prohibitive, labour intensive, and error prone, potentially resulting in project delays, and in the increase of development complexity and costs (BRESSAN; OLIVEIRA; CAMPOS, 2020; OLIVEIRA; BRAGA, et al., 2019; OLIVEIRA, André Luiz de; BRAGA, R.; MASIERO, P.; PAPADOPOULOS, et al., 2018).

Variant management solutions exist to support the management of variability in

and where a system can be used (LEE; KANG, 2010).

system design and dependability information (OLIVEIRA, André Luiz de; BRAGA, R.; MASIERO, P.; PAPADOPOULOS, et al., 2018). However, modern variability management solutions lack mechanisms for mapping variability abstractions to their materialization into finer-grained model fragments. Moreover, these techniques provide limited variability resolution mechanisms that do not guarantee the derivation of correct and complete system models enriched with dependability information concerning feature selection (BRESSAN; OLIVEIRA; CAMPOS; CAPILLA, 2021).

BVR (VASILEVSKIY et al., 2015), vXengine (HORCAS; PINTO; FUENTES, 2017), and pure::variants (PURE-SYSTEMS, 2021) techniques generate configuration models with broken dependencies that should be manually fixed, and dependability information should be manually introduced into the model after variability resolution in a clone and own fashion. Also, current industry practice for reusing dependability artefacts in certification processes relies only on the same clone & own approaches (WOLSCHKE et al., 2019; TISCHER et al., 2011).

## 1.3 PROBLEM STATEMENT

Achieving the balance between functional safety and cybersecurity assurance and reuse still remains a challenge to Software Product Line Engineering. It requires enabling support for: i) modeling variability at finer-grained safety and security information, specified as annotations of component-based system models, using failure logic and state-based modeling techniques; ii) binding variability abstractions to their realization into safety and security information ensuring the derivation of complete and ready to use models; and iii) integrating functional safety and cybersecurity analysis activities and artifacts within product line processes. Functional safety and cybersecurity analysis artefacts should be integrated within the product line core assets to enable their systematic reuse, together with other artefacts, throughout application development (WOLSCHKE et al., 2019; OLIVEIRA; BRAGA, et al., 2019; POHL; HÖCHSMANN, et al., 2018; OLIVEIRA, André Luiz de; BRAGA, R.; MASIERO, P.; PAPADOPOULOS, et al., 2018).

Extensions of conventional software product line approaches with safety analysis activities (OLIVEIRA; BRAGA, et al., 2019), and the integration of model-based safety analysis and variant management tools (BRESSAN; OLIVEIRA; CAMPOS, 2020; DOMIS; ADLER; BECKER, 2015a) have been proposed. However, the aforementioned variant management approaches focus on functional safety and reliability properties, not covering variant management on cybersecurity properties.

The diversity of threats that impact information security and functional safety posed by the higher inter-connectivity of modern and configurability safety-critical systems should be managed during the analysis of threats to cybersecurity properties, i.e., Confidentiality, Integrity, and Availability - CIA, throughout product line processes. Variation in design choices and their potential interactions may impact on a range of threats that can exploit different vulnerabilities with different consequences to information security and functional safety. So, ensuring system safety and information security becomes even more challenging (BRESSAN; OLIVEIRA; CAMPOS; CAPILLA, 2021; KENNER; MAY, et al., 2021; WOLSCHKE et al., 2019).

Research on product line engineering and security has been concerned with how configurability may reveal confidential information, harming privacy (ACHER et al., 2015), the potential security threats originating from variability bugs that could hinder source code comprehension and the uncovering of software bugs (ABAL et al., 2018), and the use of variability models to assess the threat potential of configurable systems (KENNER; DASSOW, et al., 2020). An automated approach to create a vulnerability feature model from the analysis of vulnerability databases has been proposed by (KENNER; DASSOW, et al., 2020) to support security analysts to identify the potential vulnerabilities and attack scenarios for a particular system. However, these studies lack support for managing the impact of variation in the design on both cybersecurity and safety properties.

## 1.4 RESEARCH CHALLENGES AND QUESTIONS

The integration of Model-Driven Engineering, Software Product Lines, and Safety and Security co-analysis enables the management of the diversity into dependability artefacts, but still poses the following challenges (BRESSAN; OLIVEIRA; CAMPOS; CAPILLA, 2021; KENNER; MAY, et al., 2021; BRESSAN; OLIVEIRA; CAMPOS, 2020; WOLSCHKE et al., 2019; OLIVEIRA; BRAGA, et al., 2019; OLIVEIRA, André Luiz de; BRAGA, R.; MASIERO, P.; PAPADOPOULOS, et al., 2018):

- **CH1**: Integrating model-based safety and cybersecurity co-analyses within product line processes.
- **CH2**: Reducing the gap between variability constructs and finer-grained functional safety and cybersecurity analysis information expressed as annotations attached to system models.
- **CH3**: Enabling the systematic reuse of system models enriched with functional safety and cybersecurity analysis information.
- **CH4**: Ensuring the derivation of correct and complete system models enriched with functional safety and cybersecurity information from a variability (feature) model.

The aforementioned challenges found in the state of the art of Software Variability Management techniques with respect to representing the diversity of safety and security artifacts raise the following research questions:

- **RQ1:** How to represent the diversity in system dependability artefacts?
- **RQ2:** How to derive correct and complete (with respect to feature selection) product system models enriched with dependability information from a variability model?
- **RQ3**: How model-based safety and cybersecurity co-analyses can be integrated within product lines processes?
- **RQ4:** Which software variant management technique is most effective and efficient in supporting product line engineers in managing the diversity into safety and security artefacts throughout variability realization modeling?

## 1.5 RESEARCH GOALS

The goals of this master's dissertation are developing **i)** an effective variability realization modeling language compliant with Common Variability Language (CVL) (HAUGEN; WASOWSKI; CZARNECKI, 2012) standard[4] to represent the diversity of safety and security information, **ii)** an efficient model-transformation approach to support the automatic resolution of variability into dependability artefacts that ensures the derivation of complete and correct system models enriched with safety and security information with respect to feature selection, and **iii)** a method that integrates model-based safety and security analysis within product line processes.

The variability realization modeling language and resolution tool integrated with state of the art software variant management techniques, e.g., pure::variants (PURE-SYSTEMS GMBH, 2020), BVR (HAUGEN; ØGÅRD, 2014), or FeatureIDE (MEINICKE et al., 2017), enables the specification of mappings between domain features, safety and security information throughout variability realization modeling, and automated derivation of correct and complete system models enriched with dependability information from complex product family models. The method may support engineers performing model-based safety and security analysis aware of variation in the design, and to map dependability information to domain features to make them available for reuse. Achieving the research goals demand addressing the following specific goals:

1. **Analyzing the state of the art on software variability management approaches**, e.g., CVL, BVR, and pure::variants, and variant management techniques addressing dependability, their benefits and limitations, aiming to identify the variability modeling capabilities that can be reused, adapted, or improved to enable support for mapping variability constructs to fragments of safety and security analysis artefacts throughout variability realization;

[4] CVL was approved by Object Management Group for standardization, but the process is currently frozen for legal issues.

2. **Evaluating variability resolution algorithms** from existing software variant management tools, e.g., BVR and pure::variants, in supporting the resolution of variability into MOF-compliant system models enriched with finer-grained safety and security information aiming to verify their effectiveness at resolving variability at a finer-grained level;
3. **Developing a variability realization modeling language and model-transformation approach** to support the specification of mappings between domain features and their realization into safety and security analysis and resolution of variability within dependability information specified into MOF-compliant models;
4. **Analysing the structure** of safety and cybersecurity life-cycles prescribed by cross-domain standards and implemented by model-based dependability analysis frameworks, and product line approaches to identify how model-based safety and security analysis can be integrated within product line processes;
5. **Structuring a method** that integrates model-based safety and security analysis and variant management within product line processes; and
6. **Evaluating** the feasibility of the proposed variability realization modeling language, method, and tool into case studies involving a critical system from the automotive domain.

## 1.6 SUMMARY OF CONTRIBUTIONS

In order to address the goals and answer the research questions, this dissertation introduces CRITVAR-ML modeling language, method, and tool to support the realization of domain variability into safety and security analysis artefacts, and variability resolution into re-configurable MOF-compliant system models enriched with dependability information.

CRITVAR-ML (BRESSAN; OLIVEIRA; CAMPOS; CAPILLA, 2021) is a variability realization modeling language built upon the CVL standard to support engineers specifying mappings between domain problem-space features and finer-grained safety and security information stated as annotations (elements and property values) into MOF-compliant system models (e.g., SysML, AADL) in the solution space, thus, answering **RQ1**.

The tool (BRESSAN; OLIVEIRA; CAMPOS; CAPILLA, 2021) is extensible and compatible with feature and configuration models from state-of-the-art variant management tools, e.g., pure::variants, BVR, FeatureIDE, and it supports variability resolution into MOF-compliant models enriched with safety and security analysis information, answering **RQ2**.

The proposed method (BRESSAN; OLIVEIRA; CAMPOS, 2020; BRESSAN; OLIVEIRA; CAMPOS; PAPADOPOULOS, et al., 2020) answers **RQ3** with the provision of a set of activities and tasks to support engineers performing model-based safety and security co-analyses aware of variation in the system design and its usage context, and mapping domain features to fragments of dependability artefacts to make them available for reuse.

Finally, an automotive case study was carried out to evaluate the feasibility and effectiveness of CRITVAR-ML language, method, and tool in supporting the specification of mappings to link domain features to their realization into dependability artefacts, and derivation of product-specific models with dependability information that correspond to feature selection, answering **RQ4**.

## 1.7 PUBLICATIONS

The contributions from this dissertation has been published in: a top Software Engineering conference (BRESSAN; OLIVEIRA; CAMPOS; CAPILLA, 2021), covering CRITVAR-ML and model-transformation engine; one qualified conference on Dependable Computing (BRESSAN; OLIVEIRA; CAMPOS, 2020), which introduces the method to support model-based safety analysis in variant-intensive systems; and in a Model-Based Safety Assessment conference (BRESSAN; OLIVEIRA; CAMPOS; PAPADOPOULOS, et al., 2020), which describes a method to support process tailoring in variant-intensive systems.

The contributions outside of this dissertation scope include (BRESSAN; PIOLI, et al., 2021), (GALLINA; MONTECCHI, et al., 2022), (BRESSAN; BRAGA, et al., 2020) and (BRESSAN; OLIVEIRA; CAMPOS, 2020).

## 1.8 STRUCTURE

This dissertation is organized into seven chapters. Chapter 2 presents the foundation concepts to the reader to understand the contributions of this dissertation. Chapter 3 presents an overview of related work. Chapter 4 introduces CRITVAR-ML variability realization modeling language. Chapter 5 presents the CRITVAR model transformation engine. Chapter 6 describes the process that integrates safety and security life-cycle activities into product line processes. Chapter 7 illustrates the evaluation of CRITVAR-ML language, method, and tool in the HAD vehicle automotive case study. Finally, Chapter 8 summarizes the contributions of this dissertation, their benefits, limitations, and future work.

# 2 BACKGROUND

This chapter presents an overview of the following concepts needed for the reader to understand the contributions of this work: Software Product Line Engineering; Software Variability Management; Safety-Critical Systems; System Dependability and Standards Standards; Model-based Dependability Analysis, General Purpose Languages, and CHESS Model-Based Dependability Analysis tool.

## 2.1 SOFTWARE PRODUCT LINE ENGINEERING (SPLE)

Software Product Lines comprise a collection of software-intensive systems that share a common and manageable set of features (CLEMENTS; NORTHROP, 2001). A *feature* is a distinct system characteristic visible to the end user (KANG, K. C. et al., 1998). Several cases have demonstrated the benefits of the adoption of Software Product Lines such as reduced time-to-market and costs, systematic reuse, and portfolio scalability (YOUNG; CLEMENTS, 2017). Furthermore, Software Product Lines are also known for supporting mass customization, enhancing customer satisfaction, and improving product quality and risk management (POHL; BÖCKLE; LINDEN, 2005 apud MOTTAHIR; IRSHAD; ZAFAR, 2016). Pohl, Böckle, and Linden (2005) define Software Product Line Engineering (SPLE) as "*a paradigm to develop software applications using platforms and mass customization*". SPLE comprises two phases: Domain and Application Engineering (Figure 1).

The Domain Engineering phase supports the establishment of the reusable platform i.e., defining the commonalities and variability of the product line. This phase encompasses the definition of domain requirements, design, realization, source code, test cases, and the establishment of commonalities and variability within these artifacts through a Feature Model. *Feature Models* are used to describe the commonalities and variability of a family of systems, and represent the product line variability at the highest level of abstraction (KANG, Kyo C et al., 1990).

Variability is the ability of an artifact or system to be used in different contexts by changing or customizing its characteristics and is expressed through variation points and variants (POHL; BÖCKLE; LINDEN, 2005). A variation point is a place where members of a product line may differ from each other. Such difference may be the existence of certain model artifacts (optional or alternative artifacts) or parameters. A variant is a possible instantiation of a variation point (GURP; BOSCH; SVAHNBERG, n.d.).

The commonalities and differences between products are denoted in a feature tree which, in its most simple form, nodes represent the features themselves and may present one or more children. Parent features can relate to their children features through mandatory (child feature is required), optional (child feature is optional), OR (one or more

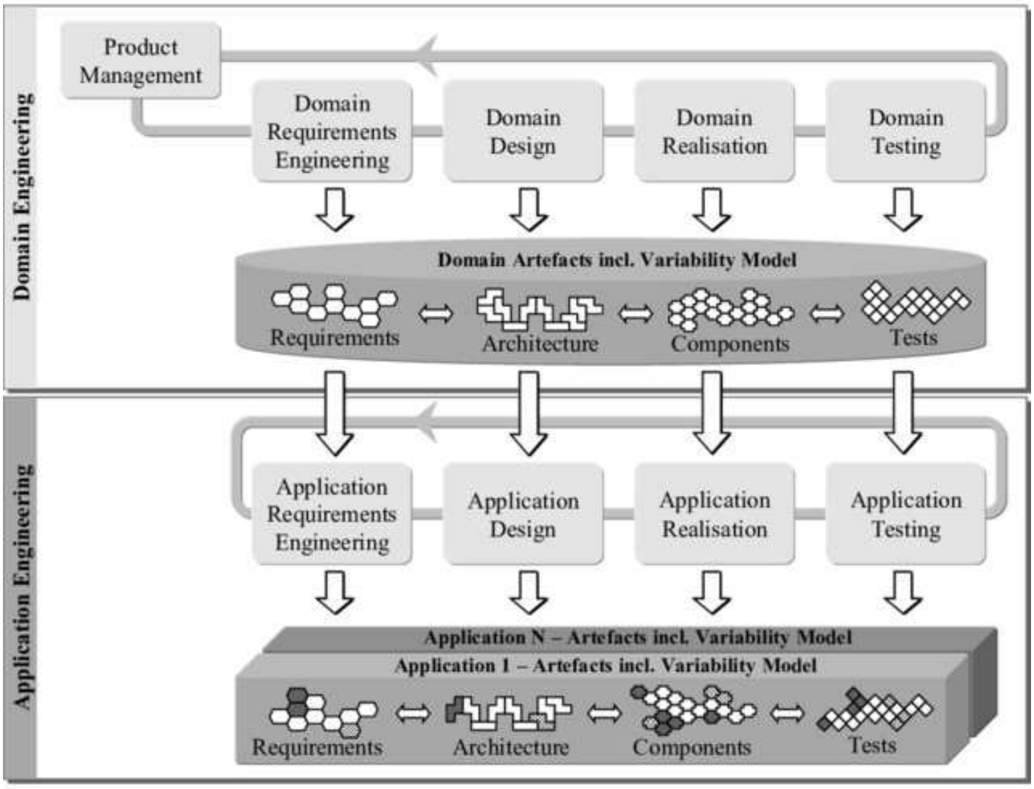


Figure 1 – Software Product Line Engineering processes (POHL; BÖCKLE; LINDEN, 2005)

child features can be selected), XOR (only one child feature must be selected) (KANG, Kyo C et al., 1990).

Figure 2 shows a feature model describing the commonalities and variability in a car Product Line. Depending on the consumer needs or application requirements, the car can implement either an Automatic or Manual transmission. Furthermore, cars with over 100hp may or may not implement an Air Conditioning system.

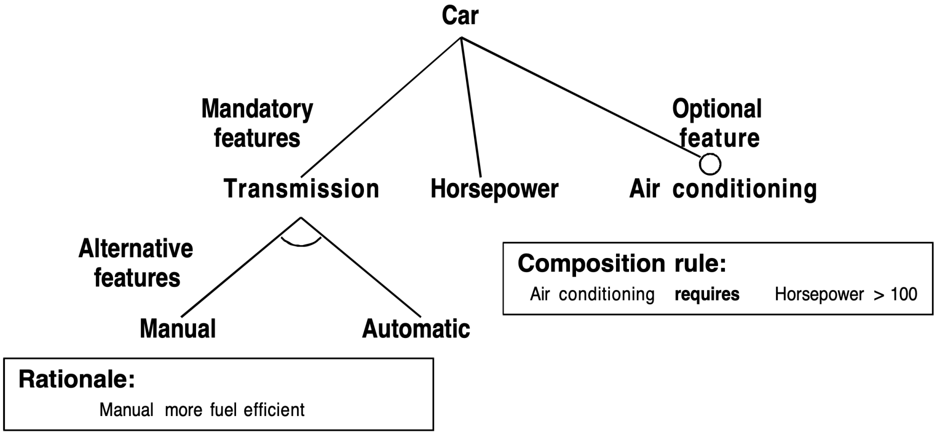


Figure 2 – Car Feature Model (KANG, Kyo C et al., 1990)

Although feature models capture the domain variation points in a concise way, their elements are merely propositional (CZARNECKI; ANTKIEWICZ, 2005 apud LACKNER; SCHLINGLOFF, 2017). The feature models only contain an abstract view of the variability domain with its commonalities and variability. The systematic and consistent reuse necessary to derive new product configurations is facilitated during the Product Management phase, by the establishment of mappings between these artifacts i.e., realization (POHL; BÖCKLE; LINDEN, 2005).

Mappings between features and artifacts can be done in two different ways: **Explicitly** through a separate variability model; or **Implicitly** within the actual referenced artifacts. Furthermore, variability modeling can be split into three major paradigms: **Annotative**, **Compositional** or **Transformational** (LACKNER; SCHLINGLOFF, 2017). In *Annotative Variability Modeling* approaches, base models store every element that is used in each possible product configuration. Variants are resolved by subtracting elements from the base model (GRÖNNINGER et al., 2014). In an *Compositional Variability Modeling* approaches, base models only contain the elements that are common among all products (GROHER; VÖLTER, 2007). Additional elements are added accordingly to resolve variants. Lastly, in *Transformational Variability Modeling*, model elements can be either added or removed to resolve a variant.

The Application Engineering phase supports the configuration i.e., resolution, and derivation, of new products. Products are built based on the reuse of domain artifacts e.g., requirements, design models and by exploiting the variability in them (POHL; BÖCKLE; LINDEN, 2005). In other words, the Application Engineering phase ensures the appropriate binding of the variability, based on each product requirements and supports the generation of different applications by reusing the domain platform artifacts.

## 2.2 SOFTWARE VARIABILITY MANAGEMENT

Software Variability Management techniques provide the realization of the Software Product Line Engineering framework, and the abstractions needed for describing and managing variability into domain artifacts. Solutions such as the Common Variability (CVL) (HAUGEN; WASOWSKI; CZARNECKI, 2012) and Base Variability Resolution (BVR) languages (HAUGEN; MØLLER-PEDERSEN, et al., 2008) provide instantiable abstractions needed throughout the Domain and Application engineering phases of SPLE, e.g., feature modeling, realization, and resolution.

The Common Variability Language (CVL) defines "*variability modeling means to generate product models*" (HAUGEN, 2014b). The Base Variability Resolution (BVR) Language comprises an evolution of CVL in which certain constructs have been removed for simplicity while others, have been added for improved expressiveness (HAUGEN; ØGÅRD, 2014). CVL/BVR are domain independent and enables the variability specification and resolution of any MOF-compliant model. Both the CVL and BVR architectures are divided into two inter-related models: Variability Abstraction (VAM) and Variability Realization (VRM) Models (Figure 3).

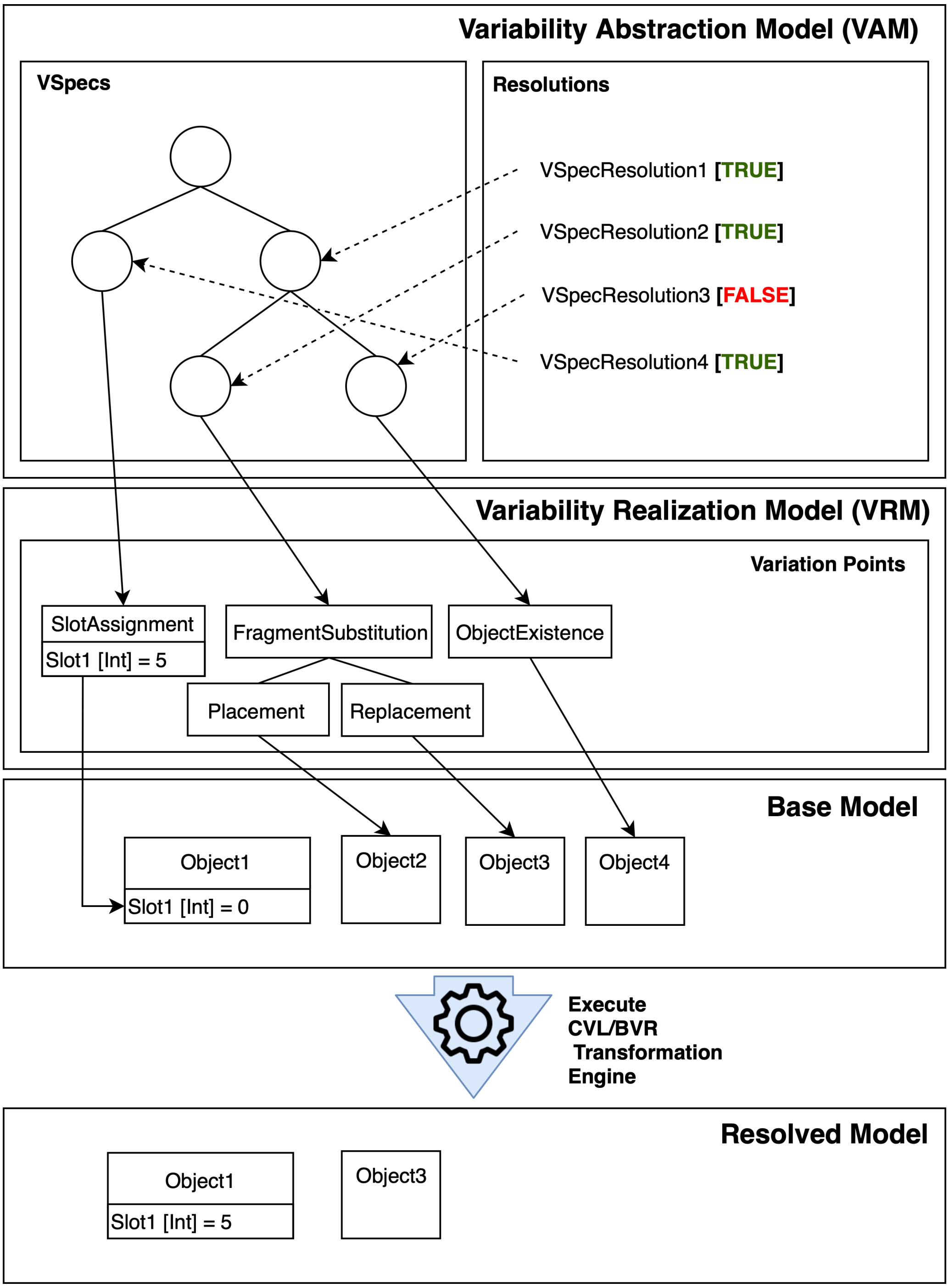


Figure 3 – CVL/BVR Architecture (adapted from Bosco et al. (2012) and Haugen and Øgård (2014)

The Variability Abstraction Model (VAM) extends traditional feature modeling and contains a tree-based structure containing Variability Specifications (VSpecs). Furthermore, the VAM also includes the resolution of VSpecs into different products e.g., if or how a certain VSpec is implemented by a certain product. VSpecs are the nodes of the variability tree and can be divided into three types: Choices, Variables and Classifiers. Similar to traditional features, Choices can be resolved to either 'yes' or 'no' e.g. by setting them as either true or false in the Resolution model. Variables are VSpecs that require a value, set in the resolution model, to be resolved. Classifiers are instantiable VSpecs that can be resolved on a per-instance basis i.e., using VInstances and setting them as true or false in the Resolution model (HAUGEN; WASOWSKI; CZARNECKI, 2012 apud BOSCO et al., 2012).

Generally, mappings between features and artifacts can be done in two different ways: *Explicitly* through a separate variability model; or *Implicitly* within the actual referenced artifacts, e.g., as done in Ziadi, Hélouët, and Jézéquel (2004) and Clauss (2001). CVL provides an *Explicit* variability mapping approach in which the links between features and artifacts are done through the Variability Realization Model (VRM). VRM is responsible for storing the mappings between VSpecs and base model elements. A *base model* is an artifact that implements the product line features e.g., UML model describing a system. Moreover, each VSpec is mapped to a fragment of the base model e.g., Classes or parameters. The VRM alongside the Resolutions in the VAM are used to specify and ensure the changes needed in resolved i.e., product-specific, models. Such changes are represented as Variation Points in the VRM and similar to VSpecs, can also be divided into three main types: Existence, Substitution and Value Assignment (HAUGEN; WASOWSKI; CZARNECKI, 2012 apud BOSCO et al., 2012). CVL *Existence variation points* can be split into two types: *ObjectExistence* and *LinkExistence. ObjectExistence* variation points are used to describe base model objects that may either exist or not across different products. Similarly, *LinkExistence* variation points, do the same for model links e.g., UML transitions.

*Substitution* variation points are also divided into two different types: *ObjectSubstitution* and *FragmentSubstitution.* Each Substitution variation point i.e., ObjectSubstitution or FragmentSubstitution contains a *Placement* and a *Replacement* fragment. Whenever a substitution variation point is activated, base model fragments referenced by a Placement are deleted and replaced by those referenced by a Replacement. *ObjectSubstitution* variation points remove and replace one single base model object while *FragmentSubstitution* variation points does the same for base model fragments containing multiple objects/links at once.

*Value Assignment* variation points are used to set values to model attributes. One of the main *Value Assignment* variation point types in CVL is *SlotAssignment.* When resolved, *SlotAssignments* allocate specific values to model slots e.g., an UML Property's defaultValue. *LinkAssignment* variation points can be used to assign model links to different objects. Lastly, *Opaque* variation points are domain-specific variation points which semantics are not defined in CVL. Their resolution behavior must be defined externally by the user according to domain specific needs.

Variation points can be classified into Compositional, Annotative and Hybrid (HORCAS; CORTIÑAS, et al., 2018). *Compositional Variation Points "define the coarse-grained variability that is applied at the architectural level"* (HORCAS; CORTIÑAS, et al., 2018) and include the traditional variation points provided in CVL and BVR, e.g., *ObjectExistence*, *FragmentSubstitution* and *SlotAssignment*. CVL is a *compositional* approach intended to be applied at a high level of abstraction, e.g., architectural, instead of working at code or extra-functional levels e.g., dependability. *Annotative Variation Points* are Opaque Variation Points, i.e., user defined, used to establish *"fine-grained variability that is applied at a lower level of abstraction"* (HORCAS; CORTIÑAS, et al., 2018). Lower level of abstraction artifacts may include source code and component dependability annotations. Annotative Variation Points semantics imply that *"there is an annotation, located in a base model artifact this variation point refers to, that is mapped to a feature". Hybrid Variation Points* share the characteristics of *Compositional* and *Annotative Variation Points*.

## 2.3 CRITICAL SYSTEMS

Sommerville (2015) defines safety-critical systems as "systems whose failure can lead to human injury or death". In addition to that, the malfunctioning behavior of critical systems may also contribute to financial losses and catastrophic consequences to the environment. Due to their critical nature, safety-critical systems are often required to have dependability properties i.e., safety and reliability, verified and demonstrated at different levels of abstraction e.g., system, hardware and software.

*Dependability* is the ability of a system to deliver its intended services in a justifiably trusted manner (AVIZIENIS; LAPRIE; BRIAN, 2000) and encompasses availability, reliability, safety, security and resilience properties (Figure 4). *Availability* describes the capability of delivering the correct service when requested. *Reliability* relates to the continuous delivery of the correct service. *Safety* is the lack of catastrophic consequences both to the user and to the environment upon the incorrect delivery of a system service and can be considered an extension of Reliability (AVIZIENIS; LAPRIE; BRIAN, 2000). *Safety* is a system property since system failures may affect the environment. *Reliability* on the other hand, is a component property and may impact on *Safety*. System Safety cannot be achieved without also achieving reliability.

*Security* comprises a composite notion that can be further broken down into availability, confidentiality and integrity. *Confidentiality* describes the absence of unauthorized information disclosure. *Integrity* relates to the ability of a system to protect itself against improper state alterations. Security is defined as the coexistence of availability, confidentiality and integrity (COMMISSION OF THE EUROPEAN COMMUNITIES, 1991)(AVIZIENIS; LAPRIE; BRIAN, 2000). Lastly, resilience describes the capability of

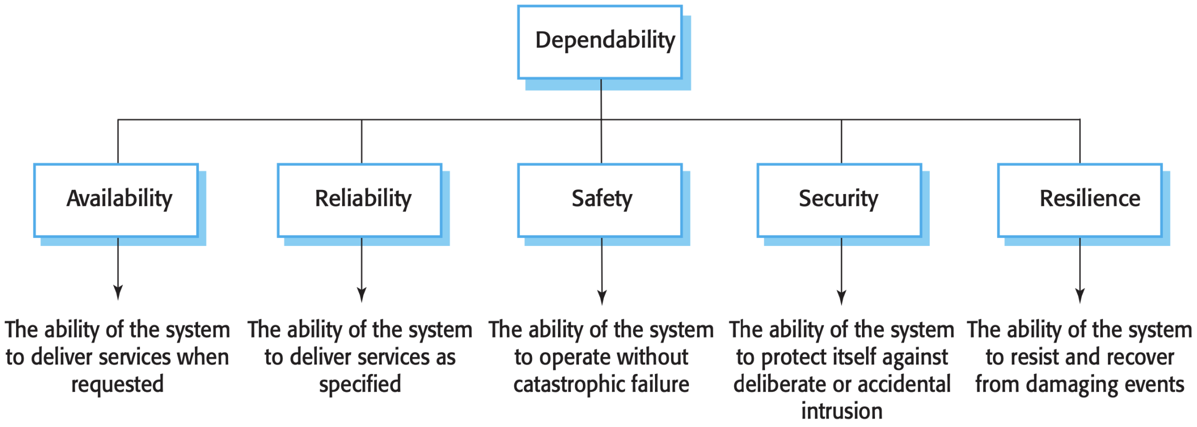

Dependability
Availability
Reliability
Safety
Security
Resilience
The ability of the system to deliver services when requested
The ability of the system to deliver services as specified
The ability of the system to operate without catastrophic failure
The ability of the system to protect itself against deliberate or accidental intrusion
The ability of the system to resist and recover from damaging events

Figure 4 – Dependability properties (SOMMERVILLE, 2015)

a system to resist and recover itself in the presence disrupting events (SOMMERVILLE, 2015).

Safety-critical systems are becoming increasingly complex and being integrated into more open and interconnected platforms. According to the National Science Foundation (2012), the term Cyber-Physical Systems (CPS) indicates the tight conjoining of and coordination between computational and physical resources. Moreover, Cyber-Physical Systems are systems that integrate aspects from both the physical and digital worlds (LISOVA; ŠLJIVO; ČAUŠEVIĆ, 2019). Due to their openness, contemporary critical systems combining both safety-critical and Cyber-Physical characteristics, must be both secure and safe (AMASS, 2018). Thus, modern critical systems are required to have their safety and security properties evaluated and demonstrated in respect to domain specific functional safety e.g., ISO 26262 (ISO, 2018), and cybersecurity standards e.g., J3061 (SAE, 2016), to ensure their proper operation and dependability (LISOVA; ŠLJIVO; ČAUŠEVIĆ, 2019).

## 2.4 SYSTEM DEPENDABILITY AND STANDARDS

Dependability Engineering is a mature discipline that was initially used as an external body responsible for analysing the causes of a system failure after an accident, and has evolved and integrated within system's development processes (BOZZANO; VILLAFIORITA, 2010). As a result of such maturity, novel safety analysis techniques have emerged and dependability practices across different domains, e.g., industry processes, automotive, aerospace, railway, medical devices, have been documented in standards throughout the last 60 years. Critical systems should be developed in compliance with safety, e.g., IEC 61508 (IEC, 2010) for electrical, electronic and programmable electronic safety related systems, ISO 26262 (ISO, 2018) for automotive and SAE ARP 4754a (SAE INTERNATIONAL, 2010) and DO-178C (RTCA, 2011) for aerospace, and cybersecurity, e.g., IEC 62443-4-1 (IEC, 2018) for industrial automation systems, ISO/SAE 21434 (ISO, 2021) for automotive, and aerospace DO-356 (RTCA, 2014), standards from the targeted domains.

Safety standards provide requirements and guidance for the analysis and demonstration of safety properties at different levels of abstraction. At the system level, engineers should identify the conditions and threats that can potentially lead to accidents, causing harm to the environment or to life, and classify their risks, e.g., using Functional Hazard Analysis (FHA) (SAE INTERNATIONAL, 1996) or HAZard and OPerability studies (HAZOP) (KLETZ, T. A, 1999; CHEMICAL INDUSTRIES ASSOCIATION, 1977). At the architectural design, the reliability of hardware components and the way how the effects of hazardous conditions propagate throughout architectural subsystems should be analyzed to identify unsafe and unreliable behaviors, and the most critical system execution paths, e.g., using top-down analysis techniques, e.g., Fault Tree Analysis (FTA) (VESELY et al., 2002), Event Tree Analysis (ETA) (ANDREWS; DUNNETT, 2000), and Petri-Nets (MALHOTRA; TRIVEDI, 1995). Finally, direct and indirect contributions of components to hazardous conditions leading to accidents should be analyzed to identify the most critical components from the architecture, and applicable safety goals and requirements to reduce dependability risks using bottom-up analysis techniques such as Failure Modes and Effects Analysis (FMEA) (US MILITARY, 1977).

The introduction of software and connectivity in modern automotive, aerospace, and systems from other domains, and the potential hazards and economic losses posed by cyberattacks increased the concern on security. Such concern is manifested both in theory (DOBAJ et al., 2019) and practice (FOSTER et al., 2015; MILLER; VALASEK, 2015). Therefore, security challenges have emerged, requiring engineering approaches and methods to deal with threats, risk management, secure design, and cybersecurity measures over the whole life-cycle (MACHER; SCHMITTNER; VELEDAR, et al., 2020). Cybersecurity standards, e.g., automotive ISO/SAE 21434, define high-level guidance for security life-cycles of cyber-physical systems, and coordination of interaction points between safety and security processes. Security life-cycles comprise Threat Analysis and Risk Assessment (TARA) to identify the potential cybersecurity threats and classify their risks, and identification of cybersecurity goals and requirements at system and functional levels to reduce security risks.

Cybersecurity analysis methods and techniques including ETSI Threat, Vulnerability, and implementation Risk Analysis (TRVA) (ETSI, 2011), Operational Critical Threat, Asset, and Vulnerability Evaluation (OCTAVE) (ALBERTS et al., 1999), HEAVENS (LAUTENBACH; ISLAM, 2014) security model, and attack trees (MOORE; ELLISON; LINGER, 2001) can be adopted to support threat analysis at the function and item levels. Besides, Cybersecurity HAZOP (SRIVATANAKUL; CLARK; POLACK, 2004), STRIDE (SWIDERSKI; SNYDER, 2004), SAHARA (MACHER; SPORER, et al., 2015), Confidentiality, Integrity, and Availability (CIA) (LIPNER; ANDERSON, 2018) analysis approaches can also support threat analysis and risk assessment, and identification of cybersecurity goals. The analysis and demonstration of dependability properties at system, design, and component levels constitute the phases of safety and cybersecurity analyses processes defined in the life-cycles of prescriptive standards.

## 2.5 MODEL-BASED DEPENDABILITY ANALYSIS

System safety and security analyses can be performed with the support of modern model-based techniques. Model-Driven Engineering (MDE) raises the level of abstraction of software specification allowing unambiguous expression of requirements and architecture, it automates system design and safety analysis. For these benefits, model-based techniques have been adopted by industry (BEUCHE; SCHULZE; DUVIGNEAU, 2016; SCHULZE; MAUERSBERGER; BEUCHE, 2013; DORDOWSKY; BRIDGES; TSCHOPE, 2011) and accepted by regulators (LISAGOR; KELLY; NIU, 2011). Moreover, standards from different domains, e.g., automotive ISO 26262 (ISO, 2011) and aerospace SAE ARP 4754a (SAE, 2010), have been recognized the maturity of model-based techniques in the development and verification of safety-critical systems. Model-based techniques (MAZZINI et al., 2016; FEILER, 2013; PAPADOPOULOS et al., 2011) allow performing safety analysis based on the preliminary system architecture and initial functional decomposition (STEWART et al., 2017). These techniques are built upon Meta-Object Facility (MOF) (OMG, 2019a), which is the OMG industry-standard for Model-Driven Engineering.

Component-based approaches such as Unified Modeling Language (UML) (OMG, 2017c) and System Modeling Language (SysML) (OMG, 2017a) from Object Management Group (OMG), Architectural Analysis and Design Language (AADL) (DELANGE, 2016), EAST-ADL (EAST-ADL ASSOCIATION, 2013), and MATLAB Simulink (MATHWORKS, 2021) are suitable for capturing system dependability information. There are several Model-Based Safety Analysis (MBSA) (JOSHI et al., 2005) frameworks that currently support engineers reasoning about faults in component-based models, such as CHESS Failure Logic Analysis (FLA) (GALLINA; JAVED, et al., 2012) and State-Based Analysis (SBA) (MONTECCHI; PURI, 2020) for Papyrus UML/SysML (ECLIPSE FOUNDATION, 2017), AADL Error Annex (FEILER, 2013) and Hierarchically-Performed Hazard Origin and Propagation Studies (HiP-HOPS) (PAPADOPOULOS et al., 2011) for AADL (DELANGE, 2016), EAST-ADL (EAST-ADL ASSOCIATION, 2013) and MATLAB Simulink (MATHWORKS, 2021) models. Extensions to CHESS, HiP-HOPS, and AADL model-based safety analysis frameworks support the analysis and verification of cybersecurity properties. These extensions support engineers reasoning about violations of CIA (ANDERSON, J., 1972) cybersecurity properties in component-based models during threat analysis and risk assessment.

Model-based dependability analysis (JOSHI et al., 2005) allow engineers to enrich component-based system models with qualitative and probabilistic functional safety, reliability, and cybersecurity properties. MBSA frameworks also enable the automatic syndissertation of fault trees, FMEA, and attack trees artefacts, which are costly to produce, from a system model enriched with dependability information. Safety is a system property related to freedom from events (accidents) that result in injuries or loss of life, significant damages to the environment or some other form of loss important to the system stakeholders (LEVESON, 1995). Reliability is the probability that a piece of equipment or component will perform its intended function satisfactorily for a prescribed time and under stipulated environmental conditions. Cybersecurity is the ability of a system to protect itself against unauthorized access to its features, data, or hardware that can compromise confidentiality[1], integrity[2], or availability[3] (LIPNER; ANDERSON, 2018; SALTZER; SCHROEDER, 1975; ANDERSON, J., 1972). Cybersecurity information relates to the potential vulnerabilities (AVIZIENIS; LAPRIE; RANDELL, et al., 2004) that can be exploited by external attacks and expose the system, potentially causing harm to its users or the environment. The sources of cybersecurity vulnerabilities can be faults in the design, e.g., unintended function interactions, development (e.g., buffer overflow), or operation (e.g., configuration error) (MCGRAW, 2006).

## 2.6 GENERAL PURPOSE LANGUAGES

This section provides an overview of the core capabilities of UML (Section **2.6.1**) and the MOF metamodeling language (Section **2.6.2**).

### 2.6.1 Unified Modeling Language (UML)

The Unified Modeling Language (UML) (OMG, 2017d) is a General Purpose Language used to visualize, specify, construct and document the artifacts of software-intensive systems (BOOCH; RUMBAUGH; JACOBSON, 2005). The abstract syntax of UML is defined through the UML metamodel. The UML metamodel is a constrained subset of itself (e.g., it uses classes to define UML Classes themselves and other UML abstractions) which is also reused in the Meta Object Facility metamodeling language specification (OMG, 2017d).

The *UML Core* (Figure 5) contains the underlying modeling concepts used in UML, e.g., Elements and Comments. The UML metaclass *Element* is the root of the *Core Package* and therefore, the superclass for all the metaclasses defined in UML. The main UML abstractions such as Classes, Relationships (e.g., Generalizations, Associations)

[1] Confidentiality: ensuring the information is only available to authorized users.

[2] Integrity: ensuring the information has not been modified by an unauthorized entity.

[3] Availability: reliable and timely access to a system and its data always when required.

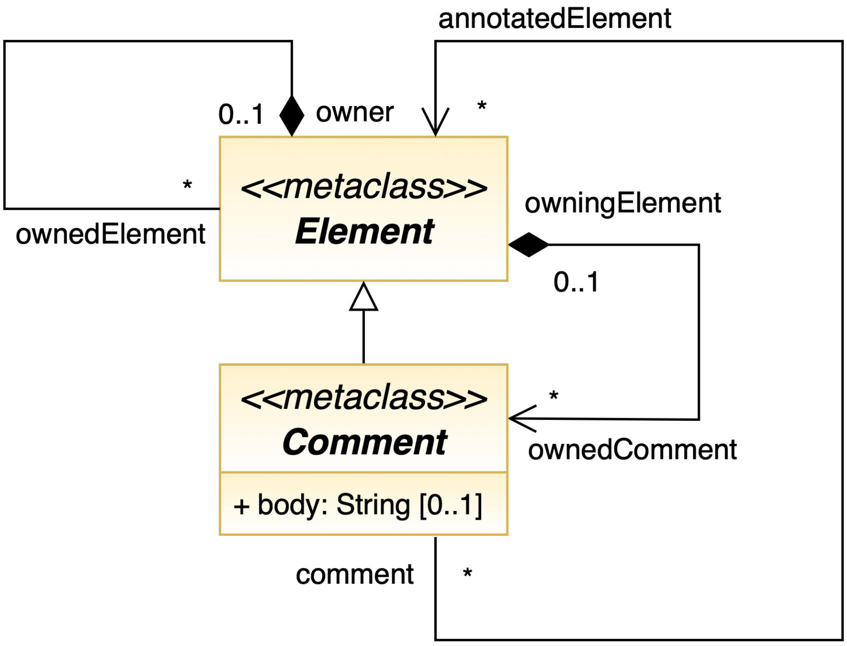


Figure 5 – Excerpt of the *UML Root* and the definition of the metaclass *Element*

and class Properties are all *Elements*. *Elements* may *own* other *Elements* (identified as *ownedElements*). The relationship between *Elements*, is defined through a composition and therefore, whenever an *Element* is deleted, all of its *ownedElements* are also removed from the model. An *Element* may also own *Comments* through the composition relationship *ownedComment*. A *Comment* is also a subclass of *Element* and provides the *body* property to which the comment text can be defined at model level i.e., M1. Furthermore, the abstractions within metamodels that use the UML Core, e.g., SysML (OMG, 2017b), are all *Elements* as well.

One of the main principles addressed by the UML metamodel is extensibility. UML provides a lightweight extensibility mechanism through *Profiles*, which enables the customization of the language and address specific platforms, solutions and domains. Additionally, UML models extended with *Profiles* are compatible with any UML modeling tool and these extensions, can be specified through the tools themselves, e.g, through Eclipse Papyrus (ECLIPSE FOUNDATION, 2017) or Capella Model-Based System Engineering (MBSE) Tool (CAPELLA, 2020). Therefore, adapting and extending UML instead of creating a MOF-based modeling language from scratch, is easier and cheaper (OSIS; DONINS, 2017). The specification of a MOF-based modeling language i.e., metamodel, requires the creation of new tools alongside the language itself (OSIS; DONINS, 2017). Examples of UML Profile extensions include the Systems Modeling Language - SysML (OMG, 2017b), a General Purpose Language (GPL), that enables the detailed specification, analysis, design and verification of systems that may include both software and hardware modules; and the CHESS Modeling Language - CHESS-ML (INTECS, 2020), a Domain

Specific Language (DSL) built upon UML/SysML. CHESS-ML contains profiles that allows the enrichment of models with dependability (e.g., safety, security and reliability) information.

Figure 6 illustrates an example of an UML Profile, its core elements and dependencies. An UML Profile diagram comprises of a *Metamodel*, i.e., a package containing the elements that are extended with the *Profile*; a reference relationship linking the *Profile* package with the *Metamodel* package; and the *Profile* package itself extending the referenced *Metamodel* through the definition of *Stereotypes*. *Stereotypes* are elements defined inside the *Profile* package that extend a *Metaclass* e.g., the *UseCase* metaclass from UML. A *Stereotype* extends existing UML vocabulary by adding a new element and describing how an existing metaclass can be extended enabling the integration of platform or domain specific terminology or notation in the modeling language (OSIS; DONINS, 2017). Stereotypes can be used to extend mataclasses with new properties called *tagged values*. *Metaclasses* are extended through the *Extension* relationship which is directed from a *Stereotype* element to the *Metaclass* it extends. At last, *Profile Application* is a dependency relationship between a package and a *Profile*, which enables the use of of *Profile Stereotypes*, in the model elements of the source package (OSIS; DONINS, 2017).

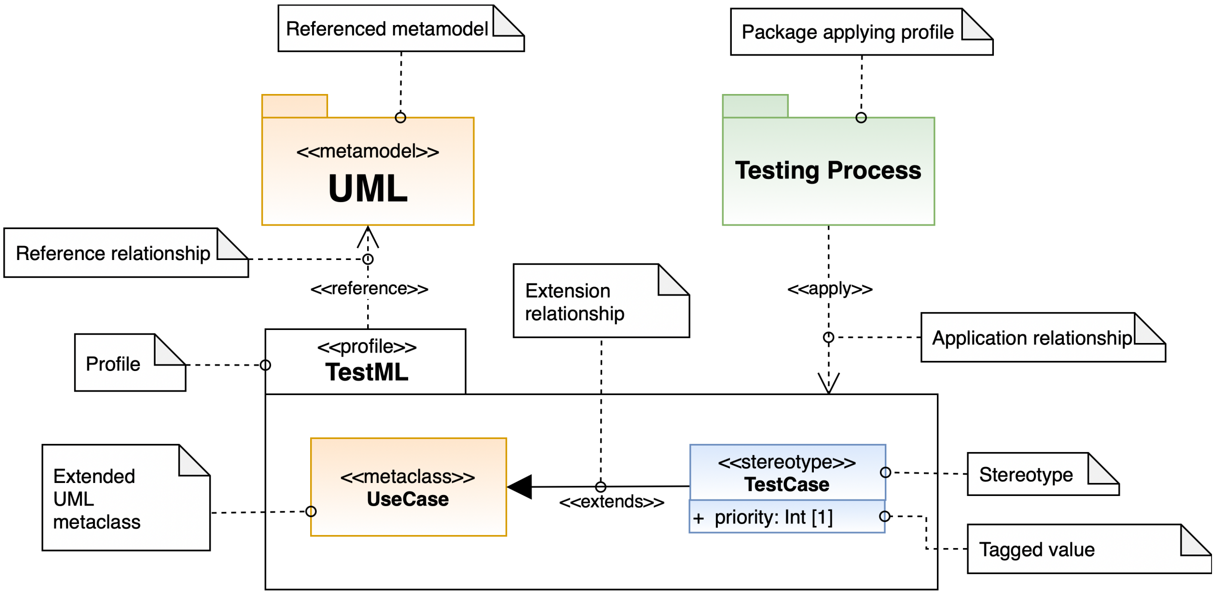


Figure 6 – Example of a profile and its application. Adapted from: Osis and Donins (2017)

### 2.6.2 Meta Object Facility (MOF)

MOF metamodeling language (OMG, 2019a) *"reuses the structural modeling capabilities of UML, based on the common metamodel shared between UML and MOF"* (OMG, 2019a). MOF comprises of two main packages: Essential MOF (EMOF) and Complete MOF (CMOF). EMOF is a subset of MOF that matches the capabilities of Object Oriented Programming Languages (e.g., Java, C++) and XML. EMOF *"is designed to match the* *capabilities of object oriented programming languages and of mappings to XMI or JMI"* (OMG, 2019a). EMOF provides a framework for mapping MOF abstractions to their implementations in XMI and Java Metadata Interface (JMI) specifications. CMOF *"provides the full metamodeling capabilities of MOF 2"* (OMG, 2019a). A similar metamodeling infrastructure and language is Ecore. Ecore provides the core meta-modeling capabilities to the Eclipse Modeling Framework (EMF). Similar to MOF, Ecore is also based on XMI and is an implementation of OMG's EMOF.

MOF uses packages to organize modeling elements into groups that can be manipulated as a set (BOOCH; RUMBAUGH; JACOBSON, 2005), used for two purposes: *importing* and *merging*. *Package importing* is a mechanism for grouping model elements together and facilitate reuse (OMG, 2019a). Package imports enable the specification of additional relationships between elements from different packages and their extension with new features (e.g, through subclassing). *Package merging* on the other hand, combines new or reusable metamodeling features and enables the creation of extended modeling languages. Once two packages are merged, classes in the *merging package* will contain all the characteristics of similarly named classes found in the *merged package* (OMG, 2019a). Unlike in package importing where elements are simply imported into the package as they are, in package merging, elements are redefined based on a concept. MOF merges the UML Core metamodel to reuse and extend its concepts, e.g., importing, subclassing, adding new classes and association between classes. Both EMOF and CMOF merge and reuse the MOF Reflection package and thus, also reuse the UML Core.

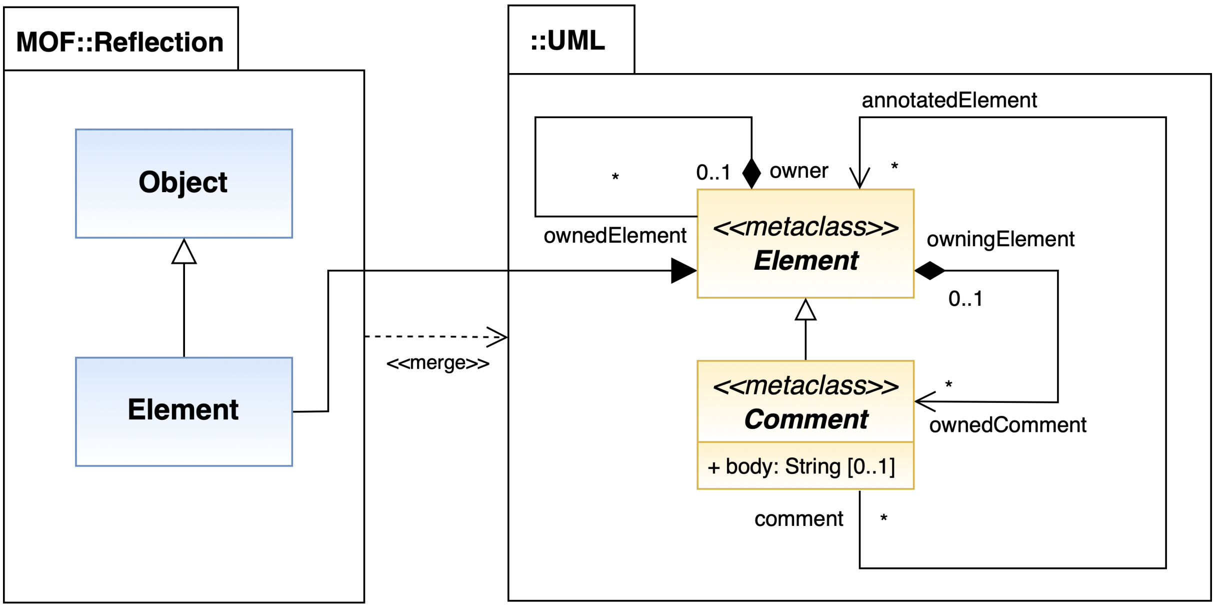


Figure 7 – Excerpt of how MOF reuses and extends the UML metamodel. Adapted from: OMG (2019a)

The underlying MOF metamodeling concepts are defined in the *Reflection Package* (Figure 7). In a MOF-based metamodel, everything is an *Object*. The *Reflection Package* enables the manipulation and the discovery of the nature of metaobjects (i.e., an object's class) and metadata. A metaobject can be used to reveal the object's kind and features (OMG, 2019a). The MOF *Reflection Package* merges the UML Core and therefore, reuses and extends many of the modeling concepts present in it e.g., classes and relationships. Just like in UML, MOF also has an *Element* metaclass which is a subclass of *Object*. The MOF *Element* metaclass merges and extends *UML::Element* and therefore, all model elements that specialize MOF Reflection's *Element*, inherit the model elements from UML (OMG, 2019a).

## 2.7 CHESS MODEL-BASED DEPENDABILITY ANALYSIS

The CHESS Modeling Language (INTECS, 2020) extends UML (through profiles) by providing abstractions to support dependability engineering activities. CHESS-ML addresses compositional compositional dependability (safety and security) analysis. Regarding safety, CHESS-ML provides abstractions for safety analysis using Failure Logic and Probabilistic Dependability Modeling. When it comes to security, CHESS-ML provides means for specifying threats, detailing attacks and their effects on safety at component level. The abstractions provided by CHESS-ML needed for enriching SysML design models with dependability information are contained in the Safety and Security profiles (both part of the CHESS-ML Dependability profile).

The *Safety* profile package defines stereotypes for specifying hazards, FPTC component annotations, component internal fault probabilities and state machines describing components' failure behavior. The *FailureMode* stereotyped element (Figure 8) can referred by system ports with the *FailureModes* stereotype i.e., through the *failureMode* tagged value, and allows the specification hazards and their risks e.g., Severity, Exposure, Controlability and Likelihood.

Figure 9 illustrates an example of the application of the *FailureMode* stereotype to denote a system level hazard and its risk. The *alarm* port with the *FailureMode* stereotype, describes the absence of a sound alert by *System A* upon experiencing a major failure. The hazard has an *exposure* of E1, i.e., very low, a *controllability* of C1, i.e., easily controllable and a *severity* of S3, i.e., possibly causing a life threatening injuries. The *likelihood* tagged value was used to assign an ASIL to the hazard which in this case is QM.

When it comes to qualitative dependability analysis and FLM, CHESS-ML also supports the enrichment of architectural elements, e.g., SysML Blocks and Parts, with FPTC annotations through the *FLABehavior* stereotype (Figure 10). Component FPTC rules are defined through the *fptc* tagged value. CHESS-ML extends the FPTC grammar defined in Wallace (2005) and requires the provision of a component port for each failure mode in the expression e.g., *[port_name.failure_mode] [port_name.failure_mode]*. Furthermore, CHESS-ML also replaces and defines two new value failure types: value-

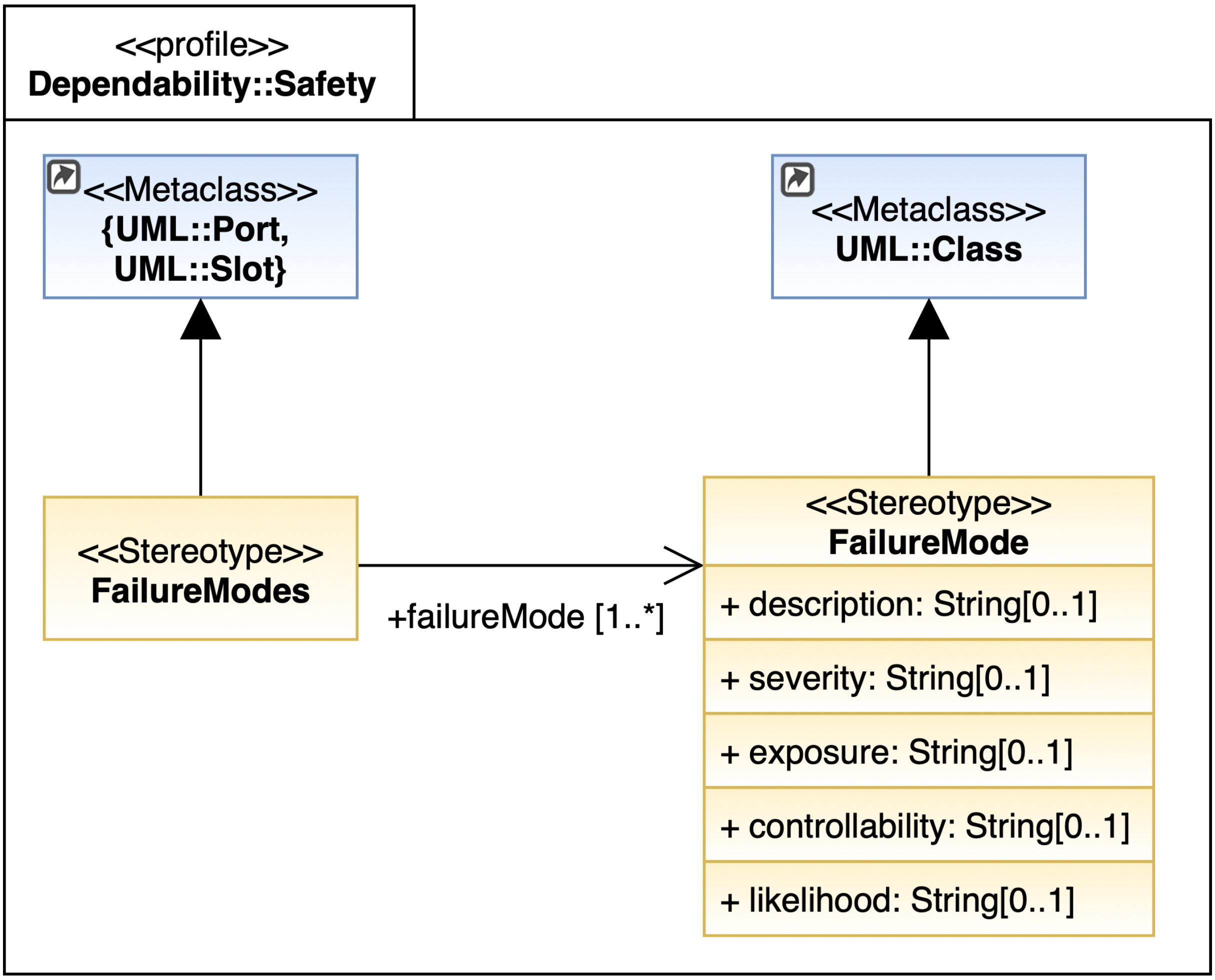


Figure 8 – The *FailureMode* Stereotype. Adapted from Intecs (2020)

Subtle and valueCoarse. *valueSubtle* denotes that an input or output has deviated from its expected value in a way which humans cannot detect. *valueCoarse* denotes that an input or output has deviated from its expected value in a way which humans can detect (GALLINA; PUNNEKKAT, 2011).

An example of component enriched with FPTC logic is illustrated on Figure 11. The *reading* output port of *ComponentA* fails with an *omission* if both *sensorA* and *sensorB* input ports fail with an *omission* as well.

The *FLABehavior* stereotype is mainly used by the CHESS Failure Logic Analysis (CHESS-FLA) plugin (GALLINA; JAVED, et al., 2012) to determine how failure modes propagate throughout the system and its components. FLA results are similar to those produced during FMEA and provide local, e.g., of a component, and system failure effects. In addition, the results obtained through CHESS-FLA can be used by the FLA2FT plugin (HAIDER; GALLINA; MORENO, 2019) to automatically derive system Fault Trees.

As for dynamic quantitative safety analysis approaches, CHESS-ML also defines a stereotype for specifying internal component failure. The *SimpleStochasticBehavior* stereotype (Figure 12) enables the specification of components' internal failure rates, (i.e.,

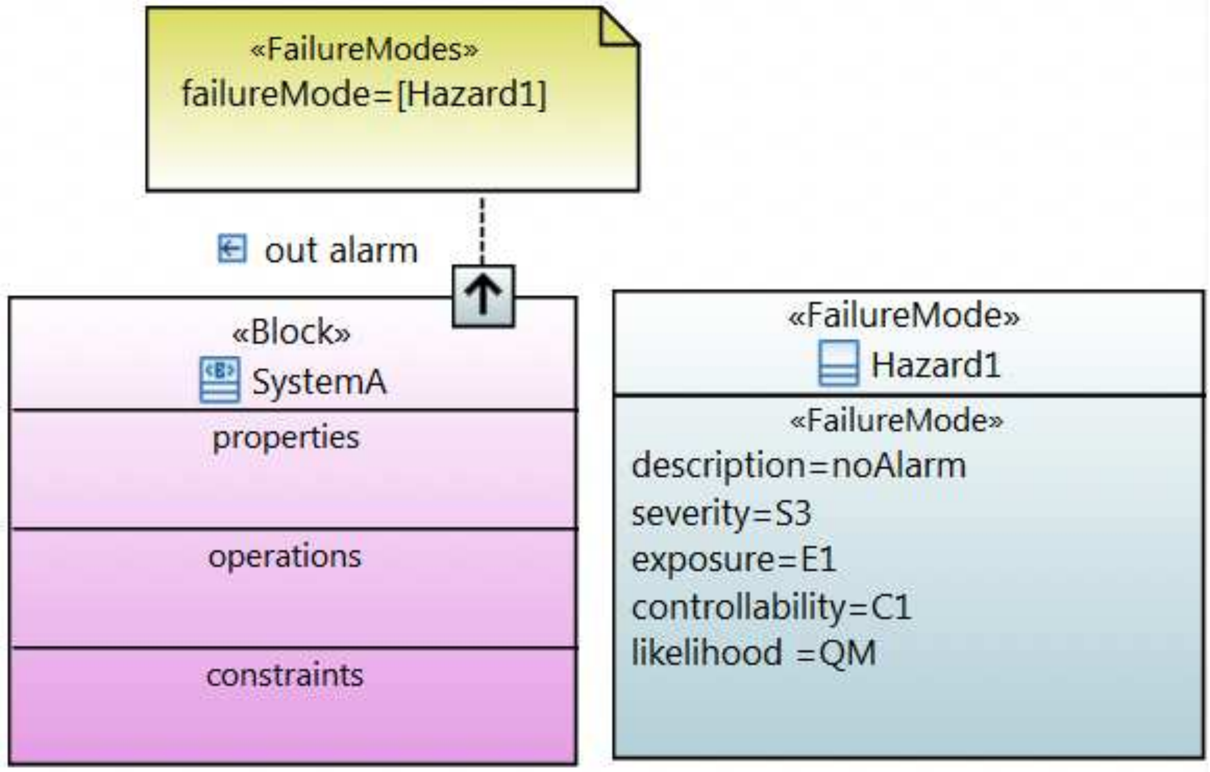


Figure 9 – Application of the *FailureMode* Stereotype

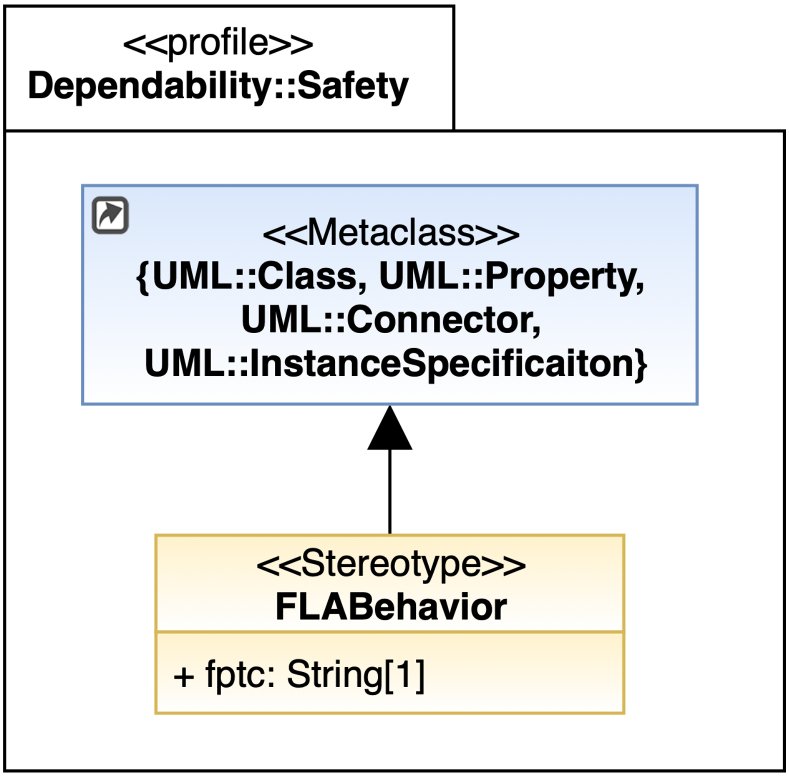
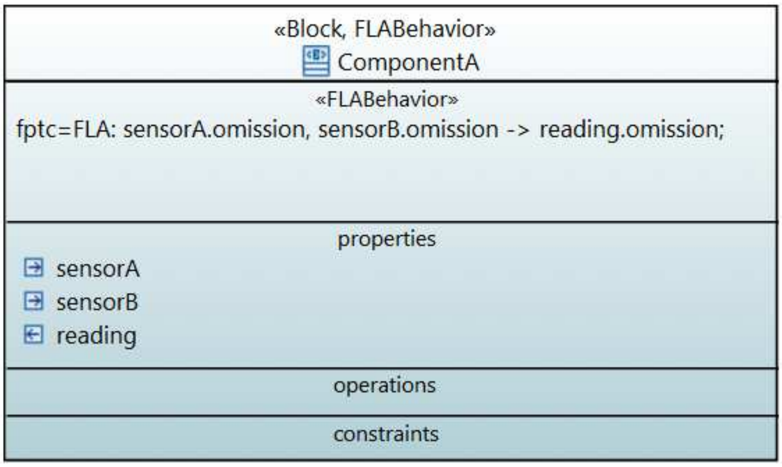


Figure 10 – The *FLABehavior* Stereotype. Adapted from Intecs (2020)

through the *failureOccurrence* tagged value), repair delay, (e.g., time a component needs to get back to its healthy state), and failure modes distribution, e.g., the failure modes propagated through the component output ports and their probabilities once an internal



«Block, FLABehavior»
ComponentA

«FLABehavior»
fptc=FLA: sensorA.omission, sensorB.omission -> reading.omission;

properties
sensorA
sensorB
reading

operations

constraints

Figure 11 – Application of the *FLABehavior* Stereotype

fault happens. The *failureOccurrence* and *repairDelay* tagged values assume a time-based probabilistic distribution, specified via the MARTE Value Specification Language (VSL) (OMG, 2019b), which can be deterministic, i.e., *det(value)*, exponential, i.e., *exp(lambda)*, uniform, i.e., *uni(alpha,beta)*, normal, i.e., *norm(mean,var)*, gamma, i.e., *gam(alpha,beta)*, and weibull, i.e., *wei(alpha,beta)*.

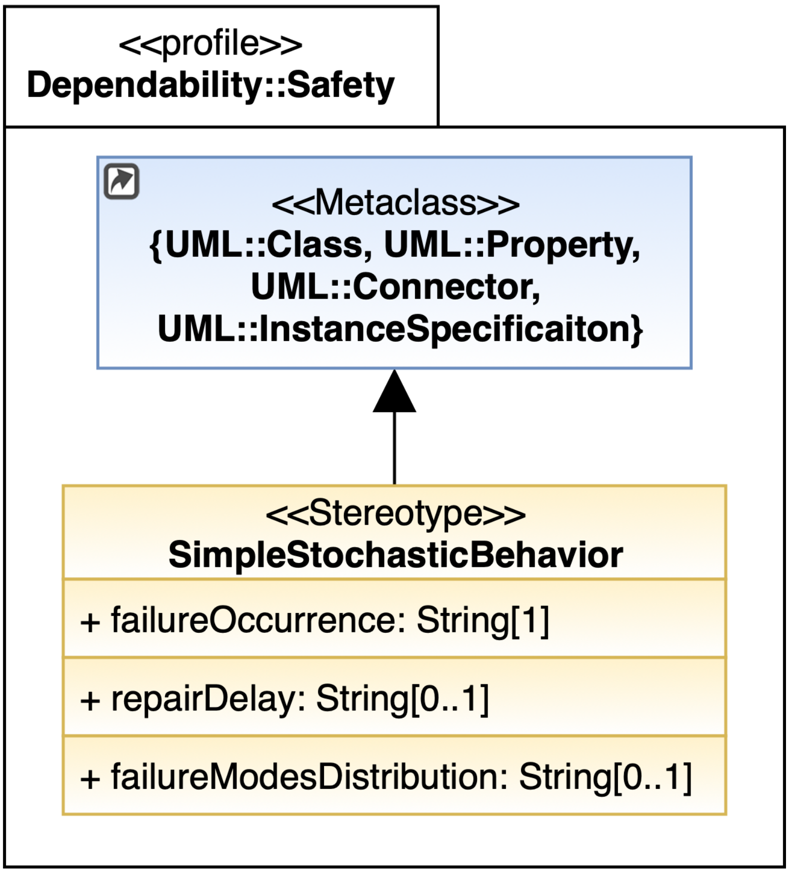
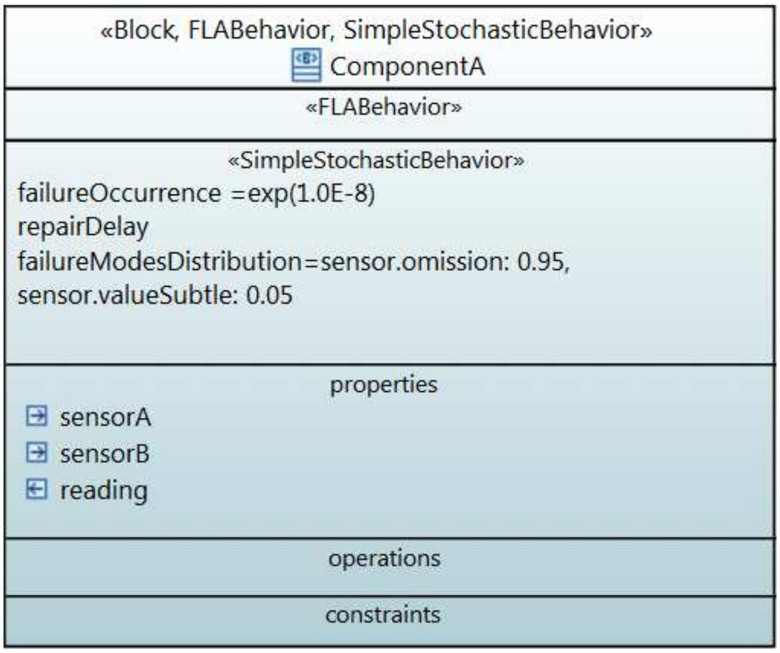

<<profile>>
Dependability::Safety
<<Metaclass>>
{UML::Class, UML::Property,
UML::Connector,
UML::InstanceSpecificaiton}
<<Stereotype>>
SimpleStochasticBehavior
+ failureOccurrence: String[1]
+ repairDelay: String[0..1]
+ failureModesDistribution: String[0..1]

Figure 12 – The *SimpleStochasticBehavior* Stereotype. Adapted from Intecs (2020)

Figure 13 shows the application of the *SimpleStochasticBehavior* stereotype on *ComponentA*. The failure rate i.e., *failureOccurrence*, of *ComponentA* follows an exponential distribution of 10^-8 per unit of time (user defined and can be in minutes, hours, days or years). Once the component fails, it stays failed and may not get back to its healthy state on its own (hence the empty *repairDelay*). Also, once failed, *ComponentA* has a 95% chance of propagating an omission and 5% of propagating a valueSuble through its *reading* output port.



«Block, FLABehavior, SimpleStochasticBehavior»
ComponentA

«FLABehavior»

«SimpleStochasticBehavior»
failureOccurrence =exp(1.0E-8)
repairDelay
failureModesDistribution=sensor.omission: 0.95,
sensor.valueSubtle: 0.05

properties
sensorA
sensorB
reading

operations

constraints

Figure 13 – Application of the *SimpleStochasticBehavior* Stereotype

Components enriched with the *FLABehavior* stereotype can also be used in conjunction with *SimpleStochasticBehavior* for computing system availability and reliability through the CHESS State Based Analysis (CHESS-SBA) plugin (MONTECCHI; PURI, 2020). Since *FLABehavior* and *SimpleStochasticBehavior* support the safety analysis of individual components in the architecture, they can also be used for manually generating either local, e.g., subsystem, array of components, or system wide Fault Trees and FMEA tables.

Another way to perform quantitative compositional dependability analysis through CHESS-SBA, is by enriching components with specialized UML state machines. Conventional UML state machines and their elements e.g., transitions and states, can also be specialized with CHESS-ML abstractions (Figure 14), to support the specification of components' error states, transitions, probabilities and delays. These state machines are specialized with the *ErrorModel* stereotype can be attached to any elements in the model that apply the *ErrorModelBehavior* stereotype.

Within *ErrorModel* state machines, states can be specialized with the *ErrorState* stereotype to denote a component's erroneous state, i.e., not complying with its specifications. *ErrorStates* can be reached through *InternalFault*, *InternalPropagation* or Failure transitions. *InternalFault* transitions describe a component's internal fault and

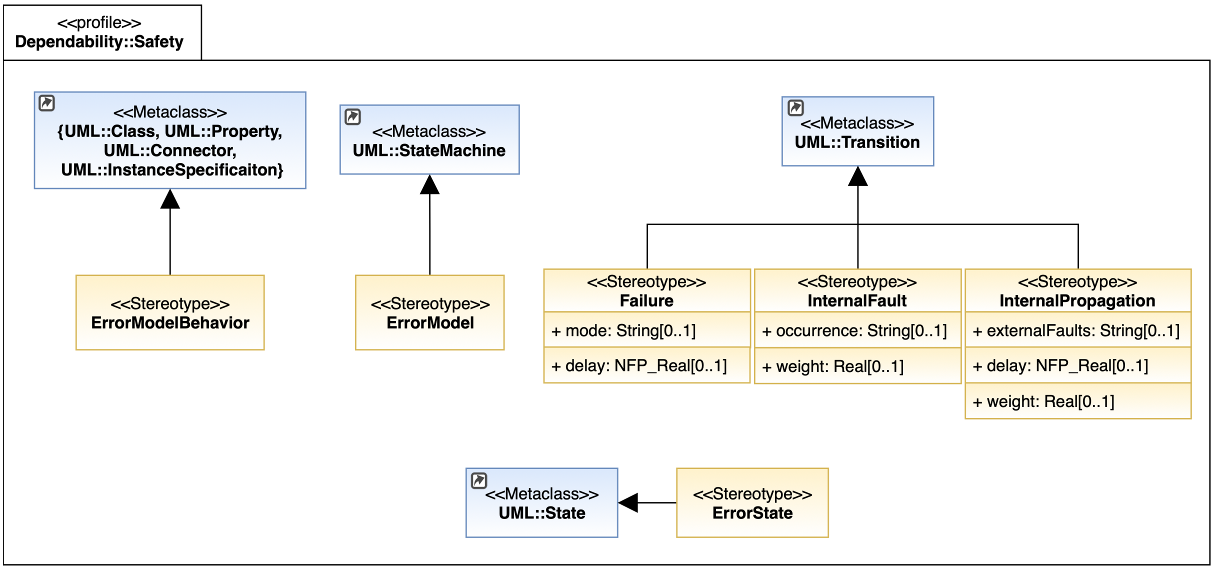


Figure 14 – CHESS-ML state machine dependability stereotypes. Adapted from Intecs (2020)

may connect either a healthy state to an *ErrorState* or two *ErrorStates* together. *InternalPropagation* transitions can be enriched with the external incoming failures that may activate them, e.g., *in1.omission AND in2.omission*, their delay, i.e., amount of time it takes for them to be activated, and weight, i.e., relative probability that a transition occurs with respect to other transitions incoming from the same state. *InternalFault* are be activated based on their *occurrence* which assumes a failure occurrence distribution based on MARTE's VSL and may also be enriched with a *weight*. Transitions that apply the *Failure* stereotype, may be enriched with the output failure modes generated by a state e.g., *out1.late, out2.late*. Models containing component *ErrorModel* state machines and components that also apply the *FLABehavior* and *SimpleStochasticBehavior* stereotypes, are transformed into Stochastic Petri Nets (SPNs) and simulated by the CHESS-SBA plugin to estimate a system's dependability proprieties, e.g., availability and reliability.

Figure 15 a) shows an example of a *ErrorModelBehavior* component enriched with an *ErrorModel* state machine (Figure 15 b). The component follows an exponential failure rate of *exp(1.0E-4)*. Once an internal failure happens, there is an 80% chance that the component will detect the failure and move to the *LateDetection ErrorState* and a 20% chance it will move to the *Undetected ErrorState*. If the component reaches the *LateDetection ErrorState* then it will produce a late failure *mode* through its *out* port. On the other hand, if the component reaches the *Undetected ErrorState* isntead, it will propagate an omission through its output port.

CHESS-ML also provides a Security Profile that supports TARA and SAHARA (MACHER; SCHMITTNER; ARMENGAUD, et al., 2017) through the analysis of the effects of attacks on safety properties, in a compositional fashion. As illustrated in Figure 16, the *Attack* stereotype is a specialized *InternalPropagation* (from the Safety Profile)

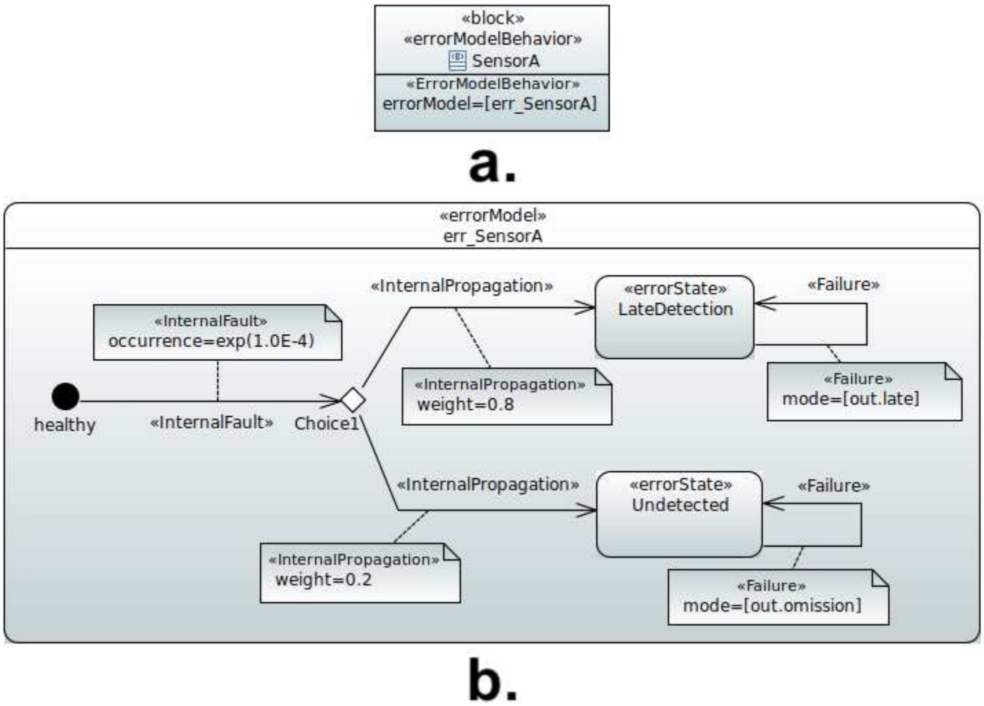


Figure 15 – Application of the *ErrorModel* Stereotype

referring to the erroneous transition caused by an external fault (an attack). It supports the specification of an attack, the threat type and exploited vulnerability, its severity, and its likelihood. An *Attack "represents an attempt to expose, modify, disable, destroy, steal or gain unauthorized access to or make unauthorized use of an asset"* (INTECS, 2020). Attacks may rise due to a *Threat* and cause a breach, if capable of successfully exploring a *Vulnerability* (INTECS, 2020). By specializing the *InternalPropagation* stereotype, *Attacks* may also be enriched with a set of *externalFaults*, (i.e., failure modes injected by the attacker into the component to try to trick or gain access to it), *delay*, (i.e., the time needed to successfully perform the attack) and *weight*, (i.e., probability that the attack is successful).

CHESS-ML allows classifying attacks into one of the following categories: *masqueradeAttack*, i.e., the attacker uses a fake identity to gain access to the system; *dataSpoofingAttack*, i.e., the attacker is able to identify itself as someone authorized to gain access or higher privileges to the system; *denialOfServiceAttack*; i.e., attack with the objective of purposely make a component or service unavailable; *bruteForceAttack*, i.e., attack based on trial and error to gain access to the system.

A *Vulnerability* represents an internal weakness of a component and may be associated with a component input port through which an *Attack* can be performed (INTECS, 2020). *Vulnerabilities* can be classified under the following categories: *missingDataIntegritySchemes* i.e., lack of mechanisms to detect and prevent attempts to modify system data, *inadequateEncryptionStrength*, i.e., lack of robust and modern encryption technologies to

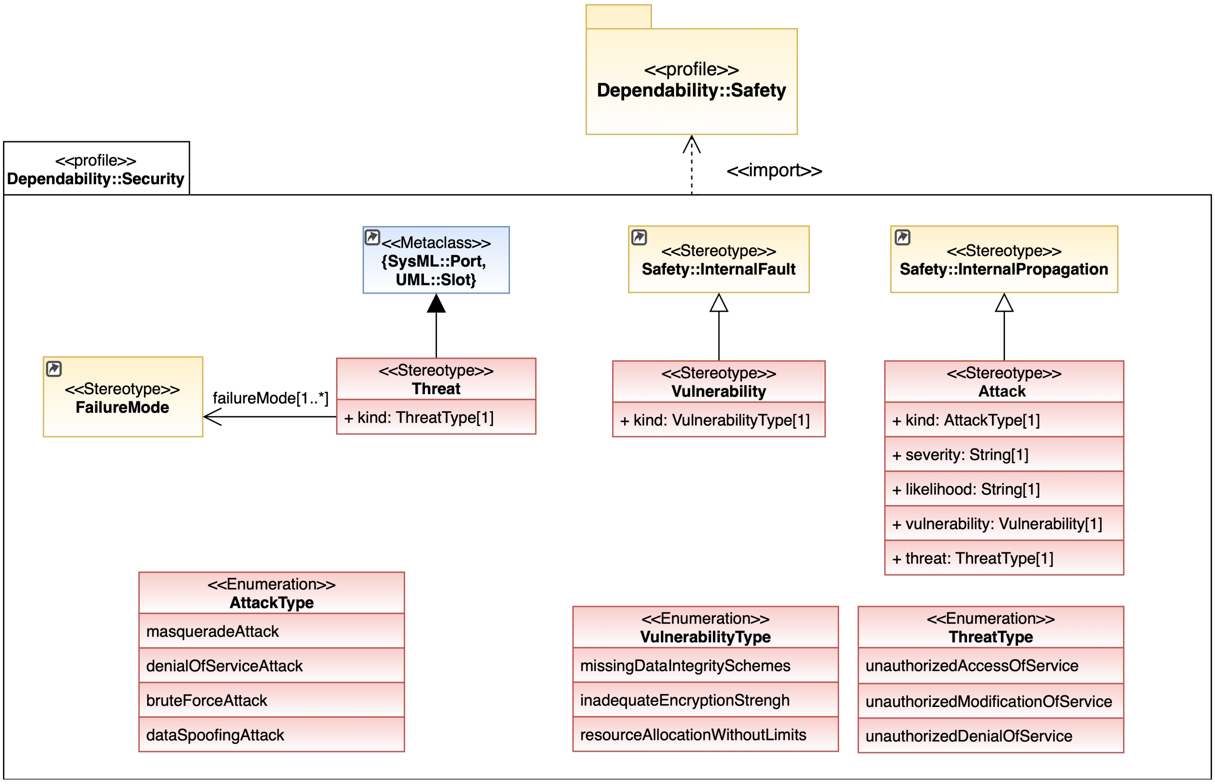


Figure 16 – CHESS-ML Security profile stereotypes. Adapted from Intecs (2020)

secure sensitive data within the system; or *resourceAllocationWithoutLimits*, i.e., absence of system resource use limitations that could potentially lead to a Denial of Service attack. Finally, attacks may also arise due to a *Threat* which can be either an *unauthorizedAccessOfService*, i.e., attacker is able to access the system without having the privileges to do so; *unauthorizedModificationOfService*, i.e., attacker is able to modify the system or its data; or *unauthorizedDenialOfService*, i.e., attacker disables the system. The effects of attacks on safety properties are established by linking *Threats* to *ErrorStates*.

Figure 17 shows an example of a state machine diagram enriched with both safety and security information. The state machine illustrates the effects of a *masqueradeAttack* that unleashes a *unauthorizedModificationOfService Threat*, by exploiting a *missingDataIntegritySchemes* vulnerability e.g., through input port *in*. Moreover, if successful, the attack generates a valueSubtle failure through the component's *reading* output port.

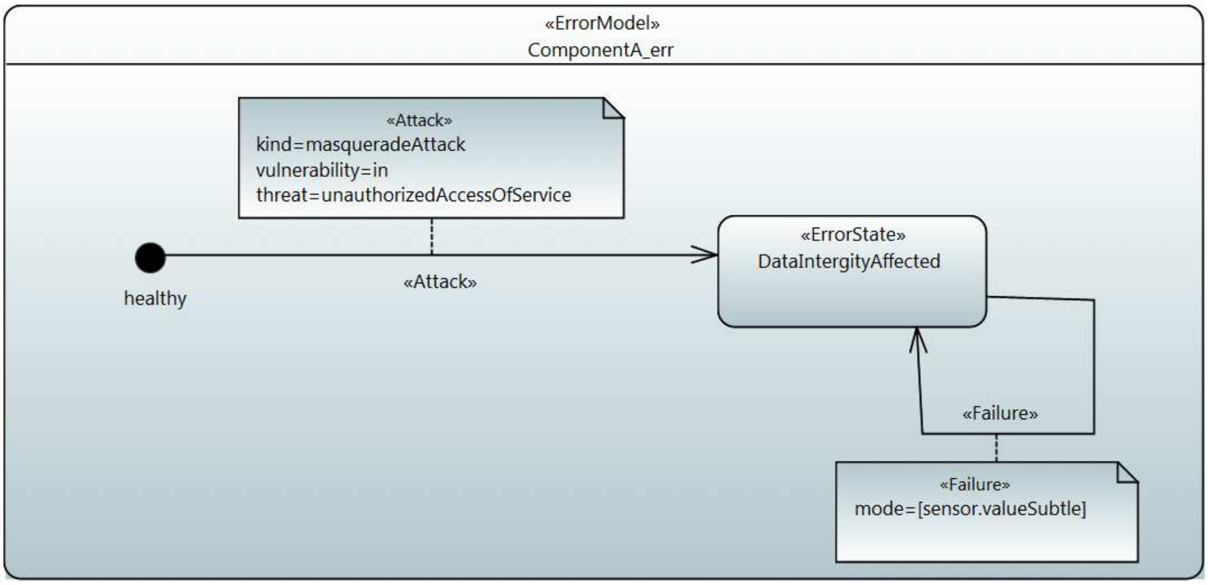


Figure 17 – Example of a component state machine containing safety and security information

# 3 RELATED WORKS

This section highlights the related works, their limitations, and research challenges addressed in this work. The state-of-the-art research on variability management in safety-critical systems covers extensions of product line processes to address system dependability properties and applications of model-based techniques to address variability in design, implementation, and dependability artifacts.

The works referenced in this section have been selected based on different strategies. Firstly, a search string has been defined with the assistance of domain experts:

(("safety" **OR** "security" **OR** "dependability") **AND** ("reuse" **OR** "software product line" **OR** “product families” **OR** "SPL" **OR** "PLE")) **AND** ("critical" **OR** "dependable" **OR** “cyber-physical” **AND** (“variability” **OR** "reconfigurable"))

The above string was used in the following research databases: Scopus, IEEEXplore, Web of Science, Science Direct, Engineering Village and SpringerLink. The following study exclusion criteria have been defined:

- **EC1:** Studies that have been written in languages other than English;
- **EC2:** Studies that are not journal, proceeding or conference papers;
- **EC3:** Studies that are set as “Preview Only”;
- **EC4:** Duplicate studies;
- **EC5:** Studies that are not related to the Software Engineering area;
- **EC6:** Secondary and tertiary studies;
- **EC7:** Studies that do not discuss or propose approaches that enable the reuse of safety or security related artifacts;
- **EC8:** Reports, case studies, evaluations, surveys.

The inclusion criteria, considered in the specified selection process was:

- **IC1:** Studies that propose approaches that enable the reuse or variability of safety/security engineering artifacts;
- **IC2:** Studies that although not being returned in the database search phase, were indicated by domain specialists.

- **IC3:** Studies referenced in papers under **IC1** that were not returned by the search string or suggested by domain specialists (snowballing).

The studies selected via the inclusion criteria were then grouped into two areas for further analysis: Extensions of Product Line Processes to Address Dependability and Model-Based Techniques to Support Variability Management of Dependability Artifacts. The selected studies are described in the following sections.

## 3.1 EXTENSIONS OF PRODUCT LINE PROCESSES TO ADDRESS DEPENDABILITY

Wolschke et al. (2019) have investigated the main issues being currently faced by the industry, regrading the reuse of safety artifacts. The authors have identified that current industry practices, rely on replicating and adapting safety artifacts with product-specific information. However, the use for "Clone and Own" strategies may lead to the inefficient reuse of artifacts and impose many challenges in the long run. Although cheap and relatively efficient in the beginning, Clone and Own strategies can eventually impact costs and require extra effort for testing, maintaining and certifying systems (SCHULZE; MAUERSBERGER; BEUCHE, 2013). Changes made in the reused artifacts are required to be fed back into the domain knowledge so that future variants are in sync with the current development environment. Thus, according to (WOLSCHKE et al., 2019), Clone and Own reuse strategies still pose challenges to the integration of product line artifacts into application development.

Engineers have also highlighted the importance of broadening the use of model-based techniques, producing generic safety information and the adoption of standardized variability specification mechanisms. Although many model-based solutions provide support for variability management, design and analysis all in a single model, traditional failure annotation and dependability analysis techniques based on Failure Logic, e.g., FPTN (FENELON; MCDERMID, 1993) and FPTC (WALLACE, 2005), and quantitative State-Based Modeling, e.g., SPNs (HAAS; SHEDLER, 1989), lack to provide support for the notion of variability. Wolschke et al. (2019) claim that simpler solutions for specific variation problems are needed since existing variability management approaches are too complex to work with. Moreover, dependability on variation points and safety artifacts should be reduced and simplified.

Notable works regarding the extension of SPL processes to address dependability include extensions to manage variability on safety analysis artifacts, e.g., FTs, (DEHLINGER; LUTZ, 2004; FENG; LUTZ, 2005) and the mapping between variant-intensive safety artifacts and state-based models (LIU; DEHLINGER; LUTZ, 2007). Dehlinger and Lutz (2004) and Feng and Lutz (2005) extend traditional FTA with SPLE phases and activities through the Product Line Software FTA approach (PL-SFTA). In domain engineering, the FT nodes are tagged with commonality or variability labels. The FTs are then be pruned accordingly and reused during the application engineering phase. The PL-SFTA approach was then extended to allow the mapping of SFTA nodes into components and state-machines representing their behavior.

Schulze, Mauersberger, and Beuche (2013) applies part of the PL-SFTA approach presented by Dehlinger and Lutz (2004) and Feng and Lutz (2005) by integrating the Ansys' Medini Analyze (ANSYS, 2020b) and pure::variants (PURE-SYSTEMS, 2021) tools to support variability management of Safety Goals, FTA and FMEA safety artifacts. The solution is built upon a referencing model, which maps problem-domain features to safety artifacts in the solution space, e.g., system FTA and FMEA results. However, the reuse of system fault trees tends to be challenging since it requires the revision of the whole tree (LISAGOR; MCDERMID; PUMFREY, 2006). The reuse of component dependability annotations tends to be more effective since variability is specified in small steps in a divide and conquer fashion. Different dependability artifacts i.e., Fault Trees, FMEA tables and reliability estimations, can be generated from one single type of dependability annotation, e.g., FPTC and state-based models.

In order to overcome the challenges of reusing system FTs, Gómez, Liggesmeyer, and Sutor (2010) propose a solution to enable the systematic reuse of Component Fault Trees (CFTs). Although adopting a compositional strategy and thus, addressing the aforementioned challenges of reusing system fault trees, the approach relies on *"clone and own"* strategies where fragments of CFTs of similar systems are reused and adapted for the specification of new CFTs. Domis, Adler, and Becker (2015b) solves that, by combining the ideas of enriching FT nodes with variability information (DEHLINGER; LUTZ, 2004; FENG; LUTZ, 2005) and systematically reusing CFTs (GÓMEZ; LIGGESMEYER; SUTOR, 2010) by integrating model-based design, and variability-management, i.e., using pure::variants, to support the reuse of Component Integrated Component Fault Trees (C2FTs) (SCHWINN; ADLER; KEMMANN, 2013). C2FTs were integrated into UML via profiles and combine Fault Trees and FMEA results into an unique artifact covering both safety analysis at component and system level. However, the approach is exclusive to graphical C2FTs notations and does not address other domain-problem space artifacts, such as probabilistic Markov models (SPNs), FPTC and Reliability Block Diagrams, that, according to Wolschke et al. (2019), still pose challenges to the reuse of dependability artifacts.

Dordowsky, Bridges, and Tschope (2011) describe their experience on implementing a helicopter SPL. The authors list a series of requirements that SPLs of the aerospace domain must follow in order to comply with guidelines to support the implementation of critical systems, e.g., DO-178C (RTCA, 2011). Such requirements include ensuring the full reusability and completeness of all components across variants without the need for modifications and guaranteeing the absence of dead code or information that do not conform to a variant, when generating source code files. However, the approach is strictly focused on the functional implementation of product line components and does not address safety analysis.

ProLiCES (BRAGA; BRANCO, et al., 2012) is an extension of traditional SPLE to address the development of safety critical systems and optimize critical software reuse. The process covers not only traditional SPLE activities such as feature modeling and resolution but also, domain and application engineering development, e.g., planning, requirements analysis, system modeling & simulation, code generation and integration, and verification activities, e.g., requirements inspection and architecture validation. In subsequent work, Braga, Trindade Junior, et al. (2012) propose a metamodel to address certification aspects, in the development of critical SPLs. The metamodel considers domain features, custom SPL process activities, e.g., ProLiCES, certification objectives and the different integrity levels product variants may achieve by including or excluding certain features. In Braga, Trindade, et al. (2012), the authors exemplify the incorporation of certification aspects into an Unmanned Aerial Vehicle (UAV) SPL. The authors present their experience on defining relationships between features and measuring the impacts of the selection of different feature combinations on integrity levels. The authors also determine the feature requirements for achieving a certain integrity level in different products.

Käßmeyer, Schulze, and Schurius (2015) propose a systematic process that integrates SPLE and Safety Engineering processes together to address the management and impacts of change onto safety artifacts. The approach is exemplified through an Advanced Driver Assistant System (ADAS) and addresses the impacts of change into HARA, e.g., risk estimation, allocation of safety goals and requirements, and Safety Analysis e.g., system and component FTs and FMEA. Unlike in approaches where variability is exclusively managed in system level solution space artifacts (SCHULZE; MAUERSBERGER; BEUCHE, 2013), the process presented by Käßmeyer, Schulze, and Schurius (2015) is incremental and addresses the mapping between variability and safety properties from the earliest stages of development. In subsequent work Kaessmeyer, Moncada, and Schurius (2015) the process was evaluated with the support of third-party model-based design, e.g., Enterprise Architect and I-SafE (now safeTBox), and variability management tools, e.g., pure::variants.

(OLIVEIRA, André Luiz de; BRAGA, R.; MASIERO, P.; PAPADOPOULOS, et al., 2018) and (OLIVEIRA; BRAGA, et al., 2019) present an extension of traditional SPLE methods to address dependability analysis during both domain and application engineering phases. Domain engineering is extended with safety engineering activities such as HARA, safety requirements and integrity level allocation, component fault modeling, and the mapping between variability information and safety artifacts. In application engineering, variants are derived and analyzed independently. Safety artifacts such as FTs and FMEA tables are then automatically generated for each variant. However, the approach is limited to safety engineering and does not address security aspects such as TARA. In addition, the approach relies on the use of third-party variability modeling, e.g., BVR, safety analysis solutions, e.g., HiP-HOPS, and therefore, is impacted by their limitations regarding the fine-grained reuse of compositional dependability information.

In conclusion, although addressing the systematic variability management in dependability artifacts, most of these approaches do not provide tool support to do so. Some approaches make use of third-party variability management and compositional dependability analysis tools such as pure::variants, BVR, and HiP-HOPS. However, these solutions lack support for either fine-grained variability specification or the generation of complete resolved models, therefore, making the systematic reuse of dependability information challenging and prone to errors. Moreover, none of the presented SPLE extensions address the systematic reuse of security information, e.g., state-based models, and the execution of security engineering tasks, e.g., TARA.

## 3.2 MODEL-BASED TECHNIQUES TO SUPPORT VARIABILITY MANAGEMENT OF DEPENDABILITY ARTIFACTS

In the category of model-based solutions to address variability, Vasilevskiy et al. (2015) propose Base Variability Resolution (BVR) a language extension of the Common Variability Language (HAUGEN; WASOWSKI; CZARNECKI, 2012) and tool, to support standard variability management on MOF (OMG, 2019a) compliant models. BVR supports Product Line Engineering Activities, i.e., feature, realization and resolution modeling, through inter-related graphical model editors. BVR also provides a model transformation engine through which, resolved models are automatically derived, according to the product requirements specified in the resolution model.

Examples of the application of BVR in critical domains include Vasilevskiy et al. (2015) and Javed, Gallina, and Carlsson (2019) for managing variability in system design, Javed and Gallina (2018) for managing variability in safety process models, and Bressan, Oliveira, and Campos (2020) for the quantitative compositional safety analysis of critical systems, through the AMASS platform. BVR has also been interated within the AMASS (VARA et al., 2019) to enable variability management in design and dependability artifacts built upon UML, SysML and CHESS-ML.

Although the benefits, BVR lacks key concepts and capabilities described in the CVL and BVR Languages that could make the systematic reuse of architectural and dependability information easier. Variability resolution is bound to Fragment Substitution operations, which consist of the removal of placement and replacement of model element fragments. The BVR Tool does not cover the resolution of variability based on *ObjectExistence*, Property Values e.g., through *SlotAssignments*, and custom implemented *OpaqueVariationPoints*. As pointed out by Horcas, Pinto, and Fuentes (2017), experiments

(DEGUEULE et al., 2015; BOSCO et al., 2012; HORCAS; PINTO; FUENTES, 2014) have shown that fragment substitutions alone are not sufficient to cover the needs of certain domains. In addition, in previous work (BRESSAN; OLIVEIRA; CAMPOS; CAPILLA, 2021), we have highlighted how the limitations of BVR makes the systematic reuse of dependability information challenging.

The BVR Tool Bundle's lack of support for the specification of variability within property values for example, makes it impossible to fully address variability in component dependability information, e.g., FPTC and failure occurrence distributions, specified as annotations in the system model. Such limitation, may lead to the derivation of incorrect critical models, i.e., that do not conform to product requirements, and imply in the adoption of clone and own strategies in the management of variability in dependability information. As a result, the reuse and analysis of variant-rich critical models, becomes a complex and error prone task.

Horcas, Pinto, and Fuentes (2017) tackles a few of the limitations of BVR regarding the lack of variability resolution methods other than *FragmentSubstitutions*, by proposing an implementation of the CVL transformation engine. vEXgine supports variability resolution based on CVL's *ObjectExistence*, and the specification of *Opaque Variation Points*. vEXgine enables the resolution of variability in UML and other MOF-compliant models by simply activating and deactivating model fragments instead replacing them through fragment substitution operations. The engine is also highly extensible and allows the specification of custom transformations through Opaque Variation Points. In previous work, (HORCAS; PINTO; FUENTES, 2014) extend CVL through Opaque Variation Points to manage variability in sequence diagrams representing security related system functions. However, the extension focuses on the functional side of security and does not address compositional security analysis. Moreover, similar to BVR, vEXgine does not support the fine-grained variability management and resolution of model properties and their values, e.g, through *SlotAssignment* variation points. Therefore, dependability annotations attached to system model elements still need to be cloned to and adapted in the resolved system model to enable the proper reuse of dependability information.

Horcas, Cortiñas, et al. (2018) proposes an approach that integrates *Annotative Variation Points* into CVL (HAUGEN; WASOWSKI; CZARNECKI, 2012) to manage fine-grained variability within web applications and their source code. The approach is hybrid and makes use of *ObjectExistence* variation points to manage variability on coarse grained artifacts, e.g., website modules, source files and components, and *Opaque Variation Points* to manage fine-grained variability at source code level. Fine-grained variability is achieved by annotating parts of the source code files with comments, instead of creating the mappings within the Variability Resolution Model like it is typically done in CVL. These comments surround portions of code, e.g., */* if (feature.[feature_name]) [variable_code] */)*, that may be deleted or kept after transformation and indicate the feature the portion it surrounds, is mapped to. Although providing means for managing fine-grained variability, the semantics behind the variation points defined in the approach, are directed towards source code and do not cover fine-grained variability in models and dependability information.

kCVL (DEGUEULE et al., 2015) is an implementation of CVL extended with the concepts and semantics needed for properly realizing patterns on MOF compliant models. kCVL introduces new variation points to CVL to support the addition of patterns into the based model, e.g., through the *PatternIntegration* variation point, and the merging between patterns and existing base model elements, e.g., via the *PatternFusion* variation point. Moreover, kCVL also ensures the correctness of the generated models by updating its elements accordingly, upon the addition, removal or integration of a new pattern. However, kCVl only covers the architectural aspects of these patterns. Similar to other CVL implementations and extensions to handle variability in MOF compliant models, kCVL also lacks support for the fine-grained variability management and reuse of extra-functional model properties such as dependability annotations.

pure::variants (PURE-SYSTEMS GMBH, 2020) is a commercial variability solution that supports variability modeling, mapping and management on Simulink and Eclipse Modeling Framework (EMF) models. The solution works mainly with the idea of *Annotative*, e.g., 150%, variability. Features are linked to elements and values through a mappings editor. Feature to model fragment mappings are established through boolean expressions, e.g., *Feature1 AND Feature2*, using the pure::variants Simple Constraint Language (pvSCl). Fine-grained model variability can be achieved through the creation of pvSCL *calculation rules* in the mappings editor. *Calculation rules* are similar to programming language functions and return a value based on the calculation's result. For example, the calculation *IF Feature1 THEN "exp(1.0E-5)" ELSE "" ENDIF* would return the string "exp(1.0E-5)" if Feature1 is activated during product configuration and an empty string case the contrary. Such returned value can be mapped to any model property or tagged value e.g., FPTC annotations, and automatically assigned during resolution. Although its benefits, pure::variants fails to fully handle the derivation of multilayered and hierarchical models such as SysML. For example, the model transformation engine fails when deleting a Block from a BDD and its Parts, i.e., instances, from an IBD. In addition, the transformation engine does not handle broken or missing dependencies in the model e.g., a connector missing its from or to ports, unless explicitly told to do so. Thus, the generated models may be incomplete since users are still required to manually delete broken dependencies from them.

Beuche, Schulze, and Duvigneau (2016) provide a solution for applying the 150% modeling approach on source code files by extending the pure::variants model transformation engine. Such a solution is based on the notion of preprocessor directives, e.g., *#if [boolean expression containing features] [source code fragment] #endif*, adopted in programming languages such as C, C++ and Assembly. Source code fragments that may vary across different variants are surrounded by these directives and a logical expression containing the features that activate them. Given a product resolution model, resolved source code files are generated by executing pure::variants model transformation engine. Although Beuche, Schulze, and Duvigneau (2016)'s approach provides means for specifying variability at a more fine-grained level, it exclusively addresses source code and its concepts have not been applied and adapted yet to work with models. We believe that integrating the ideas proposed by Beuche, Schulze, and Duvigneau (2016) and Horcas, Cortiñas, et al. (2018) and implementing them into critical models, may help us lowering the granularity with which dependability annotations can be managed and systematically reused. Thus, guaranteeing the generation of correct models and reducing the burden of dependability analysis.

Regarding variability modeling notations, several authors have proposed UML extensions via stereotypes, to enable the specification of variability within UML based models. Clauss (2001) introduces an UML extension to support feature modeling and explicit representation of variation points and optional elements in UML diagrams. Similarly, Ziadi, Hélouët, and Jézéquel (2004), proposes an UML profile for software product lines and the use of OCL to support the specification of constraints among variation points and variants. Flores and Oliveira (2021) provides a tool environment, i.e., SmartyModeling, to support the engineering of UML-based Software Product Lines. SmartyModeling implements the concepts and functionalities of Stereotype-based Management of Variability (SMarty) which include the SMartyProfile (OLIVEIRA; GIMENES; MALDONADO, 2010) and the SmartyParser (LANCELOTI et al., 2013). SMartyProfile extends abstractions defined in the UML metamodel through profiles, to enable the specification of variation points, variants and constraints as UML stereotypes. SmartyParser (LANCELOTI et al., 2013) provides a XMI parser for handling variability data provided by UML XMI-based files enriched with SMartyProfile variability annotations. None of these approaches however, provide mechanisms to define fine-grained variability.

Botterweck, Polzer, and Kowalewski (2009) propose an approach based on the Simulink variability mechanisms to represent optional (Enabler subsystems), alternative (Switch blocks), and inclusive-or (Integrator blocks) elements in Simulink models. pure::variants (PURE-SYSTEMS GMBH, 2020) provides a Simulink connector which equally enables the specification of variability using enabler, switch and integrator Simulink blocks. pure::variants simply sets these variability blocks as true or false instead of automatically removing them, alongside the affected blocks. This may generate issues during model to code transformation, since the presence of disabled blocks in the resolved models, leads to the generation of dead code. Botterweck, Polzer, and Kowalewski (2009) address this issue via an *Annotative* variability modeling and using model transformations to automatically exclude model fragments deactivated by the variability blocks.

Delta-Simulink (HABER et al., 2013) provides a different solution for variability management on Simulink models, where blocks and model fragments are encapsulated within deltas that can be added, removed, modified or replaced from the model when needed. Simulink models are then transformed into configuration models according to the selected deltas. However, none of these Similink-based variability management approaches (BOTTERWECK; POLZER; KOWALEWSKI, 2009; HABER et al., 2013) support the fine-grained variability specification of information attached to Simulink model elements. Therefore, information such as component dependability annotations still need to be cloned and adapted across different model variants.

Hephaestus (STEINER; MASIERO; BONIFÁCIO, 2013) provides support for variability management of xml-based use case scenarios, business processes, source code and Simulink models. The tool takes a feature, resolution, mappings model, (e.g., defined in a third-party tool such as pure::variants), a base model, (e.g., Simulink), and automatically generates a resolved model as a result. When applied to Simulink, Hephaestus ensures the generation of complete and concise resolved models by automatically fixing any left out broken dependencies, e.g., by reconnecting or removing broken port connectors. Hephaestus also implements fine-grained variability specification mechanisms which allows the definition of variability within data, e.g., use case descriptions. When it comes to Simulink models however, Hephaestus has only been applied to manage architectural and nominal behavior variability. Its feasibility and applicability for managing component dependability annotations is unknown. Moreover, when it comes to variability modeling in architecture and behavior, its compatibility is restricted to Simulink.

Reuling et al. (2020), adapt variability modeling concepts implemented in code-oriented SPLs and import them over to Model-based SPLs. The adapted code-oriented variability modeling concepts include Annotative variability modeling approaches, e.g., use of preprocessor directives to bound portions of code with variability with are resolved negatively, and Projectional variability modeling (BEHRINGER; PALZ; BERGER, 2017) approaches which enable the use of multiple simultaneous variability representations e.g., *Annotative* and *Transformational*, for the same code fragment. The approach presented by Reuling et al. (2020) is hybrid and allows the specification of model variability using deltas. The solution is also modeling language independent and is intended to work on any EMOF-based model. Variability is specified through EMOF Comments in the base model containing a boolean expression of feature resolutions that if satisfied, binds the appropriate variability, by deleting or adding elements to the base model, according to the chosen variability representation method. So far however, fine-grained variability has only been applied to entities such as multiplicities and property data types, e.g., by changing a property's type from Int to Float. The approach has not been yet applied to models enriched with dependability annotations and thus, further evaluation is needed to verify its feasibility for critical systems.

EPM (ABELE et al., 2012) was a prototype tool which combined system design and variability management on EAST-ADL (EAST-ADL ASSOCIATION, 2013) models. EAST-ADL provides architecture description language for describing automotive systems (BLOM et al., 2016). In EPM, variability is resolved in an Annotative (GRÖNNINGER et al., 2014) fashion, by turning base model fragments on and off. The tool also extends EAST-ADL to enable the enrichment of model components with failure annotations and hazardous scenarios.

In EPM, compositional safety analysis is supported by HiP-HOPS (PAPADOPOULOS et al., 2011), a framework that supports the automated system safety analysis in Simulink (PAPADOPOULOS et al., 2011), SysML (COLEMAN et al., 2012) and AADL (MIANI; ZARPELAO; MENDES, 2015) and EAST-ADL (WALKER et al., 2013) models. HiP-HOPS works with the idea of compositional safety analysis in which components are annotated with quantitative, i.e., failure rate distributions and probabilities, and qualitative, i.e., FPTC and FPTN based failure propagation expressions, dependability information.

HiP-HOPS summarizes the component dependability annotations and generates system wide Fault Trees and FMEA results based on user defined system hazards. HiP-HOPS also supports variability management and reuse of safety analysis information via failure model libraries. System components can be enriched with multiple implementations, i.e., different internal failure and failure propagation rules, and might be reused under different systems and contexts. Although enabling the reuse of component failure annotations, HiP-HOPS still does not fully mitigate the issues related to Clone and Own strategies. The level of granularity with which HiP-HOPS annotations can be reused is still high and engineers may still be required to repeat pieces of failure information across different component implementations. The *failure rates* of a hardware component for example, may vary depending on its manufacturer. Its generated effects however, (i.e., output failure modes), may always the same. In this case, with the component implementations strategy adopted by HiP-HOPS, engineers are required to clone and own failure annotations by replicating dependability annotation and modifying them across different component implementations.

## 3.3 DISCUSSION

From the analysis of related work, we believe that there is still space for improvement concerning the reduction of the gap between variability, dependability, and the derivation of correct critical models that conform with product requirements. Existing model variability management techniques lack to support fine-grained variability specification of model properties such as component dependability annotations, e.g., FPTC and quantitative data. Furthermore, the approaches also fail to produce complete models that conform to product requirements. On the architectural level, engineers are still required to either manually specify variability on elements that will knowingly become broken after derivation, e.g., ports and connectors, and exclude missing dependencies in the generated models for those elements that cannot be removed during model transformation e.g., Parts.

On the extra-functional level, many of the presented approaches fail to support the specification of fine-grained variability within models. This implies in the generation of incomplete models thus, requiring their manual revision and adaptation prior to system level dependability analysis. The approaches that support fine-grained variability modeling are either focused on variability specification on source code files or do not implement automated mechanisms to manage variability on dependability annotations. Therefore, engineers are required to go through a larger number of manual steps to specify variability in dependability annotations and ensure their correctness in the generated product models.

# 4 CRITVAR-ML: A MODELING LANGUAGE FOR VARIABILITY REALIZATION INTO SYSTEM DEPENDABILITY ARTEFACTS

This chapter introduces CRITVAR Modeling Language (CRITVAR-ML) to manage the diversity of safety, reliability, and security information from dependability artefacts throughout variability realization modeling, i.e., to bind domain variability constructs specified in the feature model to their realization into fragments of safety and security analysis artefacts to make them available for reuse, answering **RQ1:How to represent the diversity in system dependability artefacts**. CRITVAR-ML is an extension to the variability realization modeling concepts defined into CVL (HAUGEN; WASOWSKI; CZARNECKI, 2012) variability standard, upgraded to the BVR language (HAUGEN; ØGÅRD, 2014) by modifying the CVL metamodel with the removal of some constructs for the sake of simplicity, and addition of other constructs for more suitable expressiveness.

Section 4.1 provides an overview of CRITVAR-ML metamodel and its relationships with essential Meta-Object Facility (eMOF), UML, and CVL/BVR Object Management Group standards. Section 4.2 describes the materialization of CRITVAR-ML variability realization modeling concepts into an UML stereotype, to bind domain features to model elements, and a textual language to bind features to their realization into safety (e.g., failure mode), reliability (e.g., failure rate), security (threats or confidentiality, integrity, availability violations), or other dependability properties, e.g., repair rate, and their respective values stored into slots of elements of MOF-compliant base models. Section 4.2 describes the relationships between CRITVAR-ML concepts and Open Dependability Exchange (ODE) metamodel. Section 4.3 illustrates the integration of CRITVAR-ML and CHESS-ML to enable variant management on CHESS-FLA, Stochastic, and State-Based dependability models. Finally, Section 4.4 provides the summary of this chapter.

## 4.1 CONCEPT: CRITVAR-ML METAMODEL

CRITVAR-ML is a modeling language, which extends CVL/BVR, for specifying and resolving variability into dependability artefacts, to overcome the limitations of state-of-art variant management techniques in binding domain features to finer-grained safety and security information expressed as *MOF::Element* property values into MOF-compliant base models, and resolving variability into these models.

Figure 18 illustrates the CRITVAR-ML metamodel and its relationships with e-MOF/UML, and BVR/CVL packages. e-MOF/UML base metaclasses are highlighted in orange. CRITVAR-ML profile metaclasses are highlighted in blue. BVR/CVL variability metaclasses are highlighted in green. *e-MOF/UML::NamedElement* and *BVR/CVL::OpaqueVariationPoint* metaclasses are replicated to avoid the mixing of relationships between metaclasses from different packages. Each metaclass is detailed in the following.

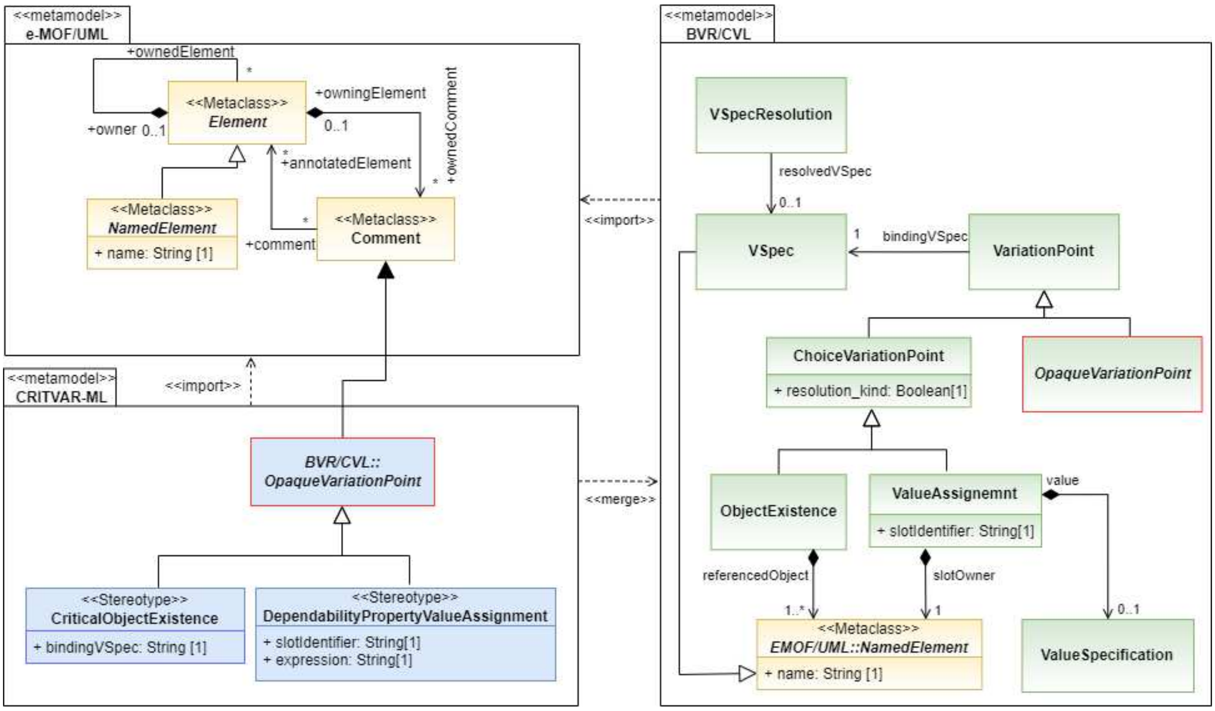


Figure 18 – CRITVAR-ML and its relationships with BVR/CVL variability and e-MOF/UML modeling concepts

**CRITVAR-ML package** metaclasses extend the *BVR/CVL::Opaque Variation Point* (*OVP*) to support custom transformations to resolve variability into *model elements* (*BVR/CVL::ObjectExistence*) and *links* between them (*BVR/CVL::LinkExistence*), and into *values* to be assigned to a *slot* (i.e., *BVR/CVL::ParametricSlotAssignment* and *BVR/CVL::ValueAssignment*) of an *element* (i.e., *BVR/CVL::ObjectHandle*) of the MOF-compliant base models.

CRITVAR-ML is a profile[1] for variability realization modeling comprising: *CriticalObjectExistence* and *DependabilityPropertyValueAssignment profile classes* (i.e., *stereotypes*), which extend *CVL::Opaque Variation Point* (*OVP*) and *e-MOF/UML::Comment* metaclasses, with properties (stereotype *tag definitions*) to support the specification of bindings between domain *Variability Specifications* (*VSpecs*) and their realization into model elements (i.e., *MOF::Elements*), element dependability properties and their values (i.e., *tagged values*) stored into *MOF::Property* slots. *CriticalObjectExistence* profile class provides a concrete implementation to *CVL::ObjectExistence* choice variation point, and *DependabilityPropertyValueAssignment* is an implementation of *ValueAssignment* variation point defined into CVL/BVR Variability Realization package (BVR/CVL package in Figure 1).

*CriticalObjectExistence* is an implementation of *CVL::ObjectExistence* choice vari-

[1] a profile class defines how an existing metaclass can be extended as part of the profile. A profile enables the use of domain specific terminology or notation in place of, or in addition to, the ones used for the extended metaclass, e.g., *e-MOF/UML::Comment*.

ation point used for binding a domain feature or a feature expression to its realization into model elements and their connections (links). If the feature or expression stated into the *VSpecs* slot evaluates true in the variability resolution model, model elements tagged with *CriticalObjectExistence* and their links are included in the product configuration model after resolution of variation points into the base model. *CriticalObjectExistence* is an implementation of *CVL::OVP* variation point, which its *VSpecs* slot is used to support the specification of a *OVP.bindingVSpec* referencing to a single *VSpec* (feature) or to an expression combining two or more VSpecs (Variability Specifications) using logical operators (AND, OR, NOT). A VSpec is a decision point that needs to be resolved, i.e., a *variation point*, and therefore, a VSpecResolution refers to a VSpec. For example, a *CriticalElementVariationPoint* UML stereotype with its *VSpecs* property set with *AutoAcceleration* domain feature, attached to *Powertrain* SysML block from a *Block Definition Diagram* denotes that the *Powertrain* block should be included into the configuration model only if the user choose *AutoAcceleration* feature in the resolution model.

The grammar below contains the rules of our VSpec logical expression language. The supported boolean operations are *and*, e.g., VSpecName1 *AND* VSpecName2 and returns True if both the VSpecResolutions are resolved to True, *or*, e.g., VSpecName1 *OR* VSpecName2 and returns True if at least one of the VSpecResolutions are resolved to True and *not*, e.g., *NOT* VSpecName1 and returns True if VSpecResolution1 is resolved to False.

⟨*expr*⟩ ::= VSpecName
 | ⟨*expr*⟩ ‘`AND`’ ⟨*expr*⟩
 | ⟨*expr*⟩ ‘`OR`’ ⟨*expr*⟩
 | ‘`NOT`’ ⟨*expr*⟩
 | ‘`(`’ ⟨*expr*⟩ ‘`)`’

*DependabilityPropertyValueAssignment* is a type of annotative variation point that indicates the existence of the entirety or portions of fine-grained dependability information, e.g., FPTC expressions and failure occurrence distributions. Dependability information that is common across different variants, must be cloned into different ValueSpecifications. Similarly, information that vary, must be adapted across the different ValueSpecifications accordingly. We tackle this issue by enabling the specification of information that is both common and variable across different variants through the *expression* parameter of the *DependabilityPropertyValueAssignment* Variation Point. Similar to how it is done by Beuche, Schulze, and Duvigneau (2016) and Horcas, Cortiñas, et al. (2018), in solutions designed to support the fine-grained variability management of source code and in pure::variants through calculations, component dependability annotations are surrounded by directives indicating the VSpecs that resolve them.

Fine-grained variability specification into component dependability annotations works in a similar way to the notion of preprocessor directives adopted in programming languages, e.g., C/C++ and Assembly. In CRITVAR, fine-grained dependability information, e.g., fptc rules and component failure occurrence probabilities, are enclosed by **#BeginVP** and **#EndVP** tags. Following each **#BeginVP** tag, comes a logical expression, containing *VSpecs*, that if satisfied , i.e., resolved as true, keeps the variable information between the *# BeginVP* and *# EndVP* directives in the resolved model. The grammar below contains the production rules needed to specify fine-grained variability using CRITVAR.

```
⟨VarExpression⟩ ::= ‘#BeginVP’ ⟨expr⟩‘:’ ⟨VarDepInformation⟩ ‘#EndVP’

⟨VarDepInformation⟩ ::= ⟨VarExpression⟩
  | fptcRule
  | failureOccurrenceProb
  | ...

⟨expr⟩ ::= VSpecName
  | ⟨expr⟩ ‘AND’ ⟨expr⟩
  | ⟨expr⟩ ‘OR’ ⟨expr⟩
  | ‘NOT’ ⟨expr⟩
  | ‘(’ ⟨expr⟩ ‘)’
```

The **BVR/CVL package** contains *VSpec*, *VSpecResolution*, *VariationPoint*, *ObjectExistence* and *ValueAssignment* elements defined into the BVR/CVL (HAUGEN, 2014b) specification for describing software variability concepts. CRITVAR-ML provides concrete implementations for *ObjectExistence* and *ValueAssignment* variation point conceptual metaclasses defined into BVR/CVL variability standard.

The **EMOF/UML package** comprises e-MOF/UML elements and their relationships. The *Element* metaclass may own properties and other elements. An *Element* may contain zero or more *Comment* annotations. A *Comment* represents a textual annotation, e.g., a remark, that can be attached to an or more *e-MOF/UML::Elements*. The *OpaqueVariationPoint* profile class extends *Comment* metaclass to support the explicit representation of variation into *e-MOF/UML::Element* and *Property* types. Essential Meta-Object Facility (e-MOF) is the OMG standard for model-driven architecture that shares a common core with UML 2.0 (OMG, 2017c). An *e-MOF::Element* extends the *UML::Element*. An *e-MOF::NamedElement* can be a *Namespace*, *ModelElement* or a *Property*. EMOF provides the basis for model-based design and safety analysis languages and tools, e.g., CHESS Papyrus UML (MAZZINI et al., 2016) and Component Fault Trees (ADLER et al., 2011) integrated within Eclipse Capella, OSATE AADL environment (CMU, 2020) with AADL Error Annex extension for dependability analysis. Since *e-MOF::Element* extends *UML::Element*, MOF elements can be enriched with variability annotations via *CriticalObjectExistence* and *DependabilityPropertyValueAssignment* profile classes from *CRITVAR-ML package*. This is possible because CRITVAR-ML profile classes extend *e-MOF/UML::Comment* metaclass, enabling both coarse-grained and fine-grained variant management on MOF *Elements*, *Properties*, and *property values*. The relationships between CRITVAR-ML and system desgin and dependability models, and implementations of CRITVAR-ML profile classes into a variability modeling language are detailed in the following.

## 4.2 CRITVAR-ML and Open Dependability Exchange Metamodel

The relationships between **CRITVAR-ML** variability modeling, **ODE::Design**, **ODE::Hazard Analysis and Risk Assessment** (HARA), and **ODE::FailureLogic** packages from Open Dependability Exchange (ODE) (DEIS, 2020) metamodel are detailed through this section. The ODE metamodel describes the model-based design and dependability (safety, security, reliability) assessment concepts shared between state-of-the-art model-based design and dependability assessment frameworks and tools, e.g., HiP-HOPS, OSATE (CMU, 2020) Architecture Analysis and Design Language (AADL) (FEILER, 2013) environment and ADDL Error Annex (DELANGE, 2016) extension for dependability analysis, the CHESS framework (MAZZINI et al., 2016), Component Fault Trees (ADLER et al., 2011) and HiP-HOPS (PAPADOPOULOS et al., 2011).

The ODE metamodel provides the core abstractions to the concept of Digital Dependability Identity (DDI) (SCHNEIDER et al., 2015) of a component or a system. A DDI contains all the information that uniquely describes the dependability characteristics of a system or components, including attributes that describe the system's or component's dependability behavior such as fault propagations, requirements on how to interact with other entities (systems or components) in a dependable way, and the level of trust and assurance. The DDI concept can be used for integrating components into systems during development as well as for the dynamic integration of systems to systems of systems in the field. The concepts embedded into ODE metamodel enable exchange and integrated analysis of modular (component) dependability-related information over supply chains.

The ODE metamodel provides concepts the represent typical models, e.g., architectural design, hazard analysis and risk assessment, and failure logic, fault trees, and FMEA required by dependability assurance lifecycles. These models can be specified with the support of model-based dependability assessment tools (PAPADOPOULOS et al., 2011; CMU, 2020; MAZZINI et al., 2016). The ODE metamodel is an exchange format for system design and dependability models produced by different tools to support different engineering stories. The ODE is a common language for exchange of models produced by different system and component vendors using different tools built upon Eclipse Modeling

Framework (EMF). EMF is a well established implementation of Model-Object Facility OMG standard for Model-Driven Engineering from which a number a tools are built. The ODE allows tool providers to implement import and export features to serialize and de-serialize (i.e., parse) data from/to their own tools.

### 4.2.1 An Overview of Open Dependability Exchange Metamodel

In the ODE metamodel, the concepts related to each lifecycle model is separated into modular packages that capture system architecture modeling and dependability analysis concepts: failure logic modeling, hazard analysis and risk assessment, and dependability requirements specification as illustrated in Figure 19. This section focuses on ODE *Base*, *Design*, *HARA*, and *Failure Logic* packages. The **ODE::Base** package encapsulates *BaseElement*, *KeyValueMap*, and *Value*, and their relationships, representing MOF *Element*, *NamedElement*, and *Property* base classes and their implementations in EMF platform. *BaseElement* is the common base class of all ODE classes in same way *Element* is the common base class of all MOF classes. A *MOF::Element* may own properties, operations, and other elements. A *ODE::BaseElement* has a unique identifier and name, and optionally a description. The elements from other ODE packages inherit *ODE::BaseElement* properties. The **ODE::Base** package also includes the *TimeUnit* enumeration commonly used in other packages.

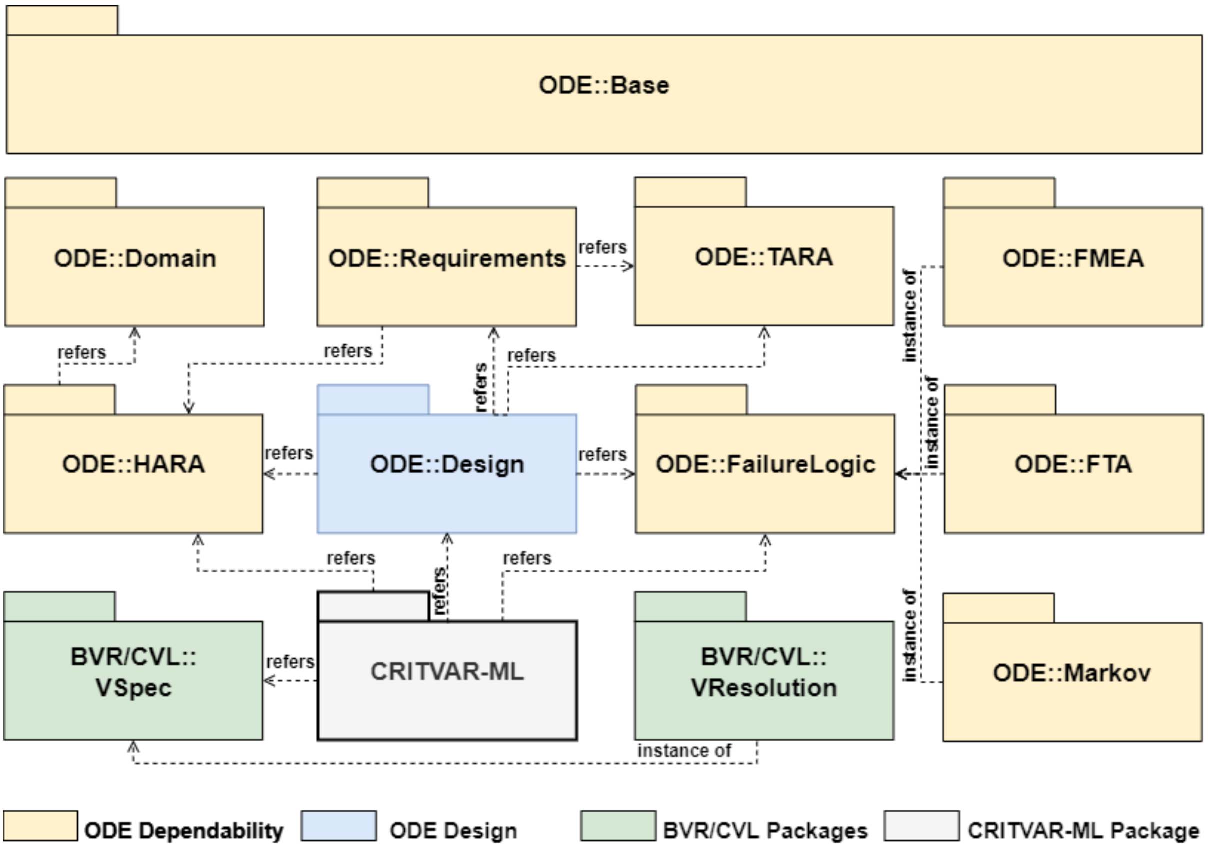


Figure 19 – Open Dependability Exchange packages and CRITVAR-ML.

Dependability can only be demonstrated in the context of structural and behavioral models of a particular system or system of systems. The concepts needed to model structural and behavioral aspects of the system architecture are encapsulated into **ODE::Design** package. In the **ODE::Design** package, the *System* can be a logical (software) or physical (hardware) representation of the system, and a *Function* is used for representing a system behavior. The *System* is the root element from **ODE::Design** package. A *ODE::System* is a composite element that may comprise subsystems and ports. *Ports* are explicitly defined interfaces from which the *System* element communicates with external systems, sub-systems, logical or physical components via signals. A *Signal* represents a connection between *Ports* thorough which the information (data) flows from a source to a destination system element. A *Port* has an assigned direction according to the direction of the signal, which can be incoming, outgoing, or both. Different phases of the engineering lifecycle focus on different aspects of the system under design such as logical, physical, and safety, which demand the analysis of different attributes. In order to support the specification and analysis of different system attributes, ODE provides *LogicalComponent*, *PhysicalComponent*, and *SafetyRelatedSystem* system sub-types. Independent from the modeling aspect, a *System* is a hierarchical representation of the architecture and has a set of *Functions* representing the behavior the system should realize (DEIS, 2020). A set of performance characteristics (*PerfChars*), i.e., non-functional requirements, can emerge from the system functional requirements and be attached to a *Function* to be realized. *System* and all design package elements inherit all basic attributes and relationships from a *BaseElement* (*ODE::Base* package).

A *System* element should be developed in compliance with dependability **ODE::Dependability::Domain::Standards**. A *Standard* provides a *Lifecycle* to develop systems in a given *Domain*, a risk-based approach for determining risk classes (i.e., *ODE::Dependability::Domain::Assurance Levels*) for classifying the risk posed by each hazard during risk assessment, and requirements for validation and confirmation measures to ensure that an acceptable level of safety is achieved. A *System* element can have an *Assurance Level* and a set of assigned *ODE::Dependability::Requirements::DependabilityRequirements* to achieve the given assurance level.

*Systems* and *Functions* can fail and their failure behavior are captured in *FailureModels* (*ODE::FailureLogic* package). A *Function* can have several *Malfunctions* (*ODE::HARA* package). A *Malfunction* describes a safety-critical deviation from the intended behavior. A *Port* can have related failure modes (*ODE::FailureLogic::FailureMode*) describing the failure propagation interface of a *System* or a *Function.* Associations between *System* and *Functions* and *ODE::FailureLogic::FailureModels*, and between *ODE::FailureLogic::Failures* and *Ports* make more explicit the relationship between system failure analysis and individual failure behaviors at the level of function/system/component. A *System* can operate within one or more specific *Contexts.* Each *Context* contains relevant information about the *System's* operation, usage, or environment. A *System* can also be present into zero or multiple *Configurations*. A *Configuration* may contain one or a set of *System* elements. Each system configuration may have different failure models and different failure propagation interfaces (*Port* failure modes).

The **ODE::HARA** package provides the modeling concepts to support hazard analysis and risk assessment, the starting point for system dependability analysis, where *Malfunctions* of the intended system behavior specified as *Functions* are identified, resulting in *Hazards* and their risks are estimated via levels of *severity*, *likelihood*, *controllability*, or other parameters associated with *RiskAssessment* element. Different *ODE::Dependability::Measures* can be assigned to eliminate or minimize the effects of a *Hazard* according to its risk. A Hazard is referenced by a *RiskAssessment* element for conducting probabilistic risk assessment to classify the risk posed by that hazard to the overall safety. A *Hazard* is also referenced by a *Function* due to the data exchange between them. *Hazard* is also referenced by a *ODE::Design::SafetyFunction* for the derivation of a safety function, and by a *ODE::Dependability::Requirements::SafetyRequirement* for the derivation of a *safety goal* and *safety requirement*. In IEC 61508 (IEC, 2010), a *safety requirement* is a requirement for a *safety function* and its associated safety integrity (assurance) levels. A *safety function* is a *function* to be implemented by a safety-related system intended to achieve or maintain the safe state for an electronic unit control with respect to a specific hazardous event. A *safety-related system* is a system aiming to implement *safety functions* to ensure the safe state for an electronic unit.

### 4.2.2 ODE::Dependability::Requirements and Failure Logic Packages

The **ODE::Dependability::Requirements** package contains the elements to model dependability requirements and their relationships with their requirement sources (*ODE::Dependability::RequirementSource*), e.g., product-specific goals and requirements, product and process requirements from relevant standards, and domain-specific regulations to achieve a given assurance level. A dependability requirement is a requirement for a system to deal with the avoidance service failures more frequent and severe than acceptable. System failure elements and their associated attributes are handled by **ODE::Dependability::HARA** and **ODE::FailureLogic** packages.

A dependability requirement can be assigned to mitigate the effects of one or more failures (ODE::FailureLogic::Failure). An assigned dependability requirement may require applicable measures (*ODE::Dependability::Measure* and maintenance procedures (*ODE::Dependability::MaintenanceProcedure*) to address a specific assurance level (*ODE:: Dependability::AssuranceLevel*). A safety requirement is a sub-type of dependability requirement derived from hazards (ODE::HARA::Hazard) to mitigate their effects on the overall system safety. A security requirement is a kind of requirement referencing a security capability to ensure system and/or data confidentiality, integrity, and availability. A measure is an applicable action or mechanism, e.g., fault avoidance and fault removal, to reduce the risks to system dependability. A fault tolerance measure is a risk reduction measure, identified during functional hazard analysis, and used to derive safety goals and safety requirements. A maintenance procedure is an applicable measure to reduce the risks of failures in physical (hardware) components.

FTA, FMEA, and Markov models lead to dependability measures (ODE::Dependability::Measure to mitigate hazard and failure effects. The *DependabilityRequirement* references an *ODE::FailureLogic::Failure* and a *ODE::Dependability::Measure*. A safety requirement can be assigned to mitigate the effects of one or more hazards. A *ODE::TARA::Security Requirement* element represents a requirement from security domain associated with a number of *ODE::TARA::SecurityCapabilities*, e.g., authentication, authorization.

Safety analysis is performed based on the top-level dependability requirements derived from hazards identified during hazard analysis and risk assessment describing the potential causes of failures in a *System* or *Function* leading to hazards to be mitigated (DEIS, 2020). The **ODE::FailureLogic** package and its sub-packages contain the elements to model the potential system failures and their causes using existing safety analysis techniques such as Fault Tree Analysis (FTA) (VESELY et al., 2002), Failure Mode and Effects Analysis (FMEA) (US MILITARY, 1977), and probabilistic Markov modeling (RABINER; JUANG, 1986). The **ODE::FailureLogic** package (Figure 2) and its sub-packages provide the elements to support modular and hierarchical failure analysis based on the system design using analysis techniques such as CHESS Failure Logic Analysis (GALLINA; JAVED, et al., 2012), HiP-HOPS (ADLER et al., 2011), and AADL Error Annex (DELANGE, 2016). The *Failure* element abstracts common characteristics of failures within functions, systems, or components.

Failures in architectural elements can be categorized into input, output, or internal according to its origin as illustrated in Figure 20. This abstraction is useful for composing failure analysis results of hierarchical models. The failure behaviors of a system or component is encapsulated into one or more *FailureModels*. A *Failure* can be a Common Cause Failure (CCF), when its occurrence can trigger other *Failures*. A probabilistic value for a failure rate can be assigned to *PhysicalComponent* (hardware) failures. A probability distribution (*ProbDist*) model can be used to calculate the unavailability of a hardware element for a given failure. There are a number of probability distribution failure models, e.g., Constant Failure and Repair Rate, Mean Time to Failure and Repair (MTTF and MTTR), with different formulae and parameters to calculate the unavailability of a component for a given basic event (failure).

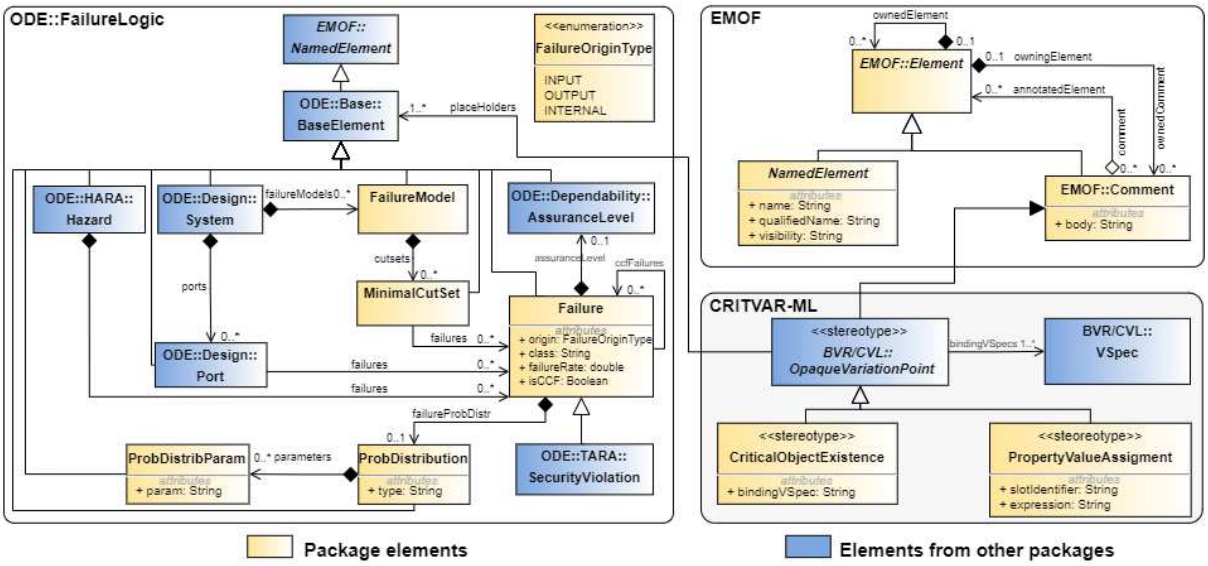


Figure 20 – ODE failure logic, EMOF, and CRITVAR-ML packages.

A *Failure* is characterized by its *class* attribute, used to describe its nature, e.g., data *omission* or *commission*, incorrect *value*, data sent too *early* or too *late* for safety analysis. An *Assurance Level* can be assigned to classify the risk posed by a software failure to system dependability. A component *FailureModel* may have a set of minimal cut-sets. A minimal cut-set represents the smallest possible combinations of *Failures* leading to the occurrence of a top-event, i.e., an output failure mode. Minimal cut-sets are used as the basis for failure propagation and probabilistic analyses. An input or output failure is associated with a given system or component port. A hazard can be associated with several output failures.

### 4.2.3 ODE::FailureLogic Sub-packages

The **ODE::FailureLogic::FTA** sub-package captures the information produced during FTA. *FaulTree* is a *FailureModel* comprising a set of *Cause* elements. A *Cause* element can be an input, output, or a basic event, or a Gate of a fault tree. Cause references a *Failure*. A *Gate* is Boolean logic event connector, which is a subtype of *Cause*, used to chain hierarchies of causes in a fault tree.

The **ODE::FailureLogic::FMEA** sub-package captures the information produced during FMEA. The *FMEA* element is a failure model comprising a set of entries (*FMEAEntry*). Each *FMEAEntry* relates an output failure from a system element under analysis with its system level effects. The FMEA element can be a FMEA or a FMEDA element. A FMEDA element has a set of *FMEDAEntries*. *FMEDAEntry* specializes *FMEAEntry* with a *diagnosisRate* double attribute, and a relationship with *ODE::FailureLogic::ProbDist*, used for calculating the system unavailability for the referenced failure mode.

The **ODE::FailureLogic::Markov** sub-package provides the elements to support the analysis of dynamic and temporal aspects of the system behavior from Markov analysis technique. A Markov model comprises a set of normal and fail *States*, with one of them set as the initial state, and probabilistic *Transitions* between states. A *State* references a failure element from *ODE::FailureLogic* package. A *Transition* element represents a transition from a source state to a destination state. Each state *Transition* has a probabilistic attribute, calculated using a probability distribution (*ProbDist*) model.

### 4.2.4 ODE::Threat Analysis and Risk Assessment Package

The **ODE::TARA** package contains the elements to support security Threat Analysis and Risk Assessment (TARA). In a TARA model, a *ThreatAgent* can be either a *Human* or *NonHuman* source of an *Attack*. An attack may serve to a purpose. A threat agent is a mean to achieve the overall goal of the attacker (*AttackerGoal*), negatively impacting *Assets*, e.g., system operation and its data, being considered for security. An attack exploits a system *Vulnerability.* A vulnerability is a weakness or error in the system source code, when exploited, may lead to a *SecurityViolation*, compromising confidentiality, availability, or data integrity through unauthorized access, elevation of privileges, denial of service, or other security threats (MICROSOFT, 2009).

An exploited system vulnerability may potentially result in a *SecurityViolation*, impacting safety and other dependability properties. *SecurityViolation* is a *Failure* sub-type for modeling the direct effect of the occurrence of a security threat (*ODE::TARA::Attack*), identified during TARA, on the overall system dependability. *ODE::FailureLogic::Security Violation* (see Figure 20) is used for modeling and linking an individual *Attack* to the propagation of its effects on the system dependability. This link enables hybrid security-safety co-analysis since complex *ODE::FTA::Causes* can also be associated with *ODE::FailureLogic:: Failure* (Figure 20) elements from safety, reliability, robustness, availability, and other dependability analysis. *SecurityCapabilities* and *SecurityControls* are high-level and low-level applicable security counter-measures to reduce the effects of security threats (*attacks*) and their risks. Security capabilities are associated with security requirements (*ODE::Dependability::Requirements::SecurityRequirements*). Security controls are associated with *ODE::Dependability::Measures.*

The CRITVAR-ML provides elements to support the specification of mappings (bindings) between domain features stored into BVR/CVL::Variability Specification models (feature models) to their realization into ODE model elements, properties, and property values. Figure 20 illustrates the relationships between **CRITVAR-ML**, and ODE **Design**, **HARA**, and **Failure Logic** elements. Since *ODE::Base::BaseElement* and all other ODE model elements are sub-types of *EMOF::NamedElement*, CRITVAR-ML variation point elements, which inherit from *BVR/CVL::OpaqueVariationPoint* can be used to bind domain features and/or feature expressions to their realization into ODE model elements and specific values for model element properties via *bindingVSpecs* and *placeholders* relationships.

### 4.2.5 CRITVAR-ML Variation Points and ODE Design and Failure Logic Packages

*CRITVAR::CriticalObjectExistence* variation point can be used to map a single feature or a feature expression (via *bindingVSpec* property) to its realization into one or more ODE **Design**, **HARA**, **FailureLogic**, and **TARA** model elements. Therefore, it is possible to map features to their realization into *ODE::Design::System* elements, *ODE::HARA::Hazard* and *ODE::FailureLogic::Failure* safety analysis elements, and *ODE:TARA::Threat* and *ODE::FailureLogic::SecurityViolation* security analysis elements, and probabilistic *ODE::FailureLogic::ProbDistribParam* elements as illustrated in Figure 20.

For example, a *CriticalObjectExistence* variation point can be used to map *FeatureA* to *S1* and *S2* objects from *ODE::Design::System* element, their ports and connections as shown in Figure 21. Another *CriticalObjectExistence* variation point can be used to map *FeatureB* to its realization into *S2* system object. Therefore, when *FeatureA* is chosen in the *BVR/CVL::Variability Resolution* model, *S1* and *S2* system elements, their ports and the connection between *S1* and its output port should be included into the product ODE::Design model after product derivation process (see Figure 21). On the other hand, only *S2* system object should be included into the final product when *FeatureB* is selected.

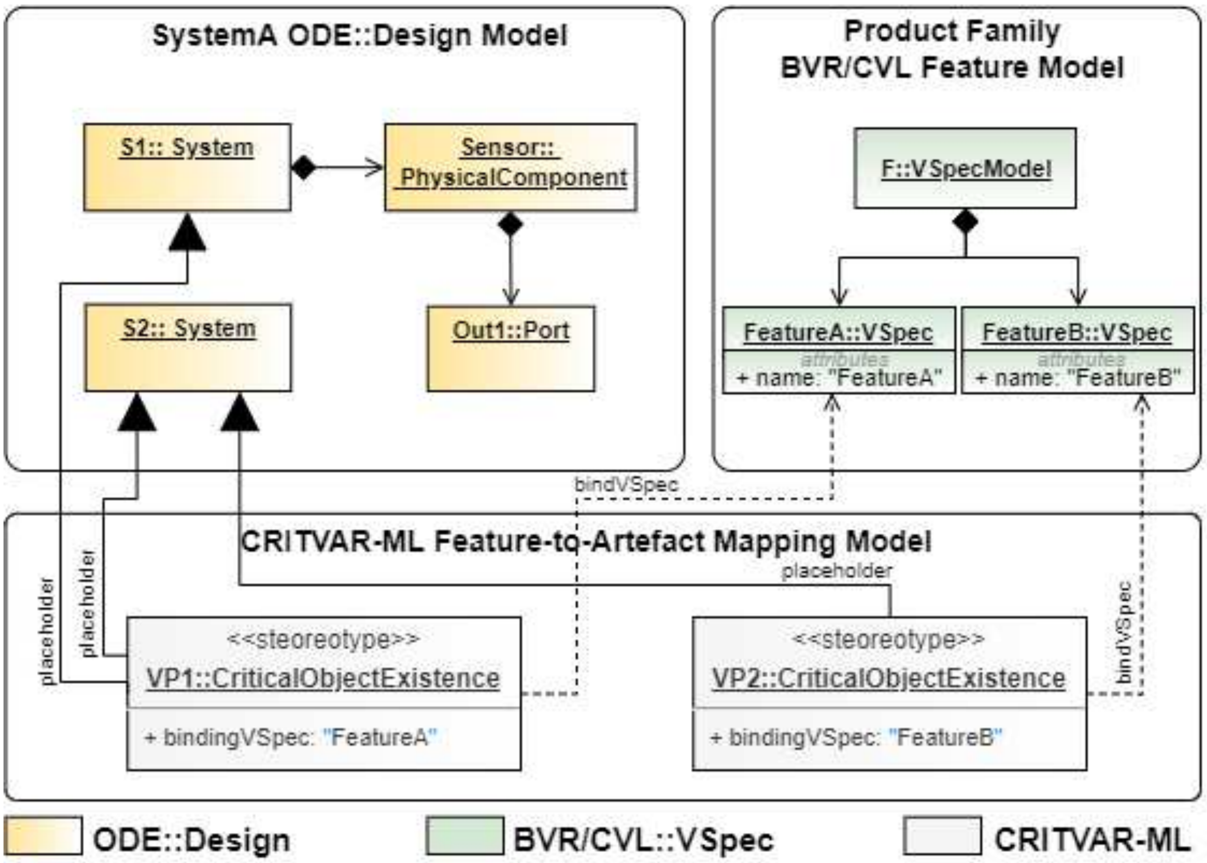


Figure 21 – Variant managament in ODE design elements using CRITVAR-ML element variation point.

*CRITVAR-ML::PropertyValueAssignment* variation point can be used to map features to their realization into specific values for a given ODE *Design*, *HARA*, *FailureLogic*, and *TARA* model element property, enabling variant management at a finer-grained level in both design, safety and security analysis elements. Figure 22 the usage of *CRITVAR-ML PropertyValueAssignment* variation point *VP3 object* to map features of a feature model to their realization into specific values for the failureRate property from a *F1 Failure* object type associated with the *Out1* port from a *Sensor* physical component. The *failureRate* property from *F1* is a placeholder referenced in the *slotIdentifier* property from *VP3 PropertyValueAssignment.*

Different values can be assigned to the *failureRate* property from *F1* failure element according to the chosen feature defined in the product family feature model (*F VSpecModel*). When *FeatureA* is selected, the value *10e-9* should be assigned to *F1 failureRate* property. On the other hand, 10e-6 value should be assigned to the *failureRate* property when *FeatureB* is chosen. The bindings between features and property values are specified via logical expressions stated in the *expression* property from *VP3* variation point object ( Figure 22). Each **BeginVP EndVP** expression block binds a *VSpec* feature or a feature expression to a specific value for the *failure rate* placeholder property from *F1* failure object.

Since *ODE::FailureLogic::SecurityViolation* extends *ODE::FailureLogic::Failure* element, it is possible to map features to its realization into a specific security violation element using *CRITVAR-ML::CriticalObjectExistence* variation point. *CRITVAR-ML::PropertyValueAssignment* can be used to map a single feature or feature expression to its realization into a specific value for a given security violation property, e.g., *failureRate*, *class*, *origin*. In this dissertation, a grammar was implemented to support the specification of **BeginVP EndVP** feature-property value expression mappings stored into *expression* properties from *PropertyValueAssignment* variation points (see Section 3.3).

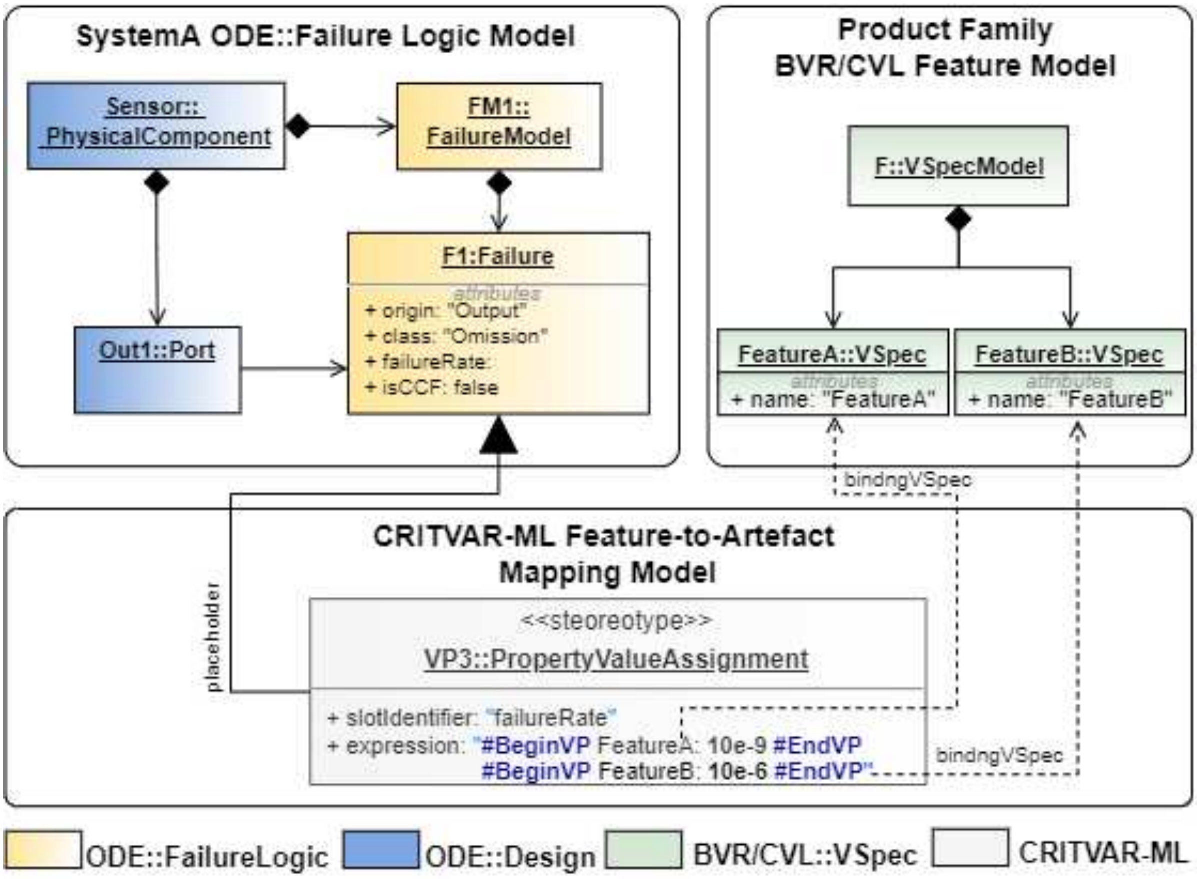


Figure 22 – Variant management in ODE failure logic property using CRITVAR-ML variation point.

## 4.3 MATERIALIZATION OF CRITVAR-ML INTO CHESS-ML MODELS

This section presents the materialization of CRITVAR-ML, to manage and resolve structural and parametric variability into CHESS-ML models enriched with functional safety, reliability, and security information (see Figure 23). CRITVAR-ML variation points support the specification of mappings between feature and feature expressions specified into CVL (HAUGEN; WASOWSKI; CZARNECKI, 2012) compliant feature models, e.g., BVR (VASILEVSKIY et al., 2015) *VSpec*, pure::variants (PURE-SYSTEMS, 2021) *.vdm* models, to their realization into CHESS ML model elements and their dependability annotations specified using CHESS-FLA and CHESS-SBA plugins.

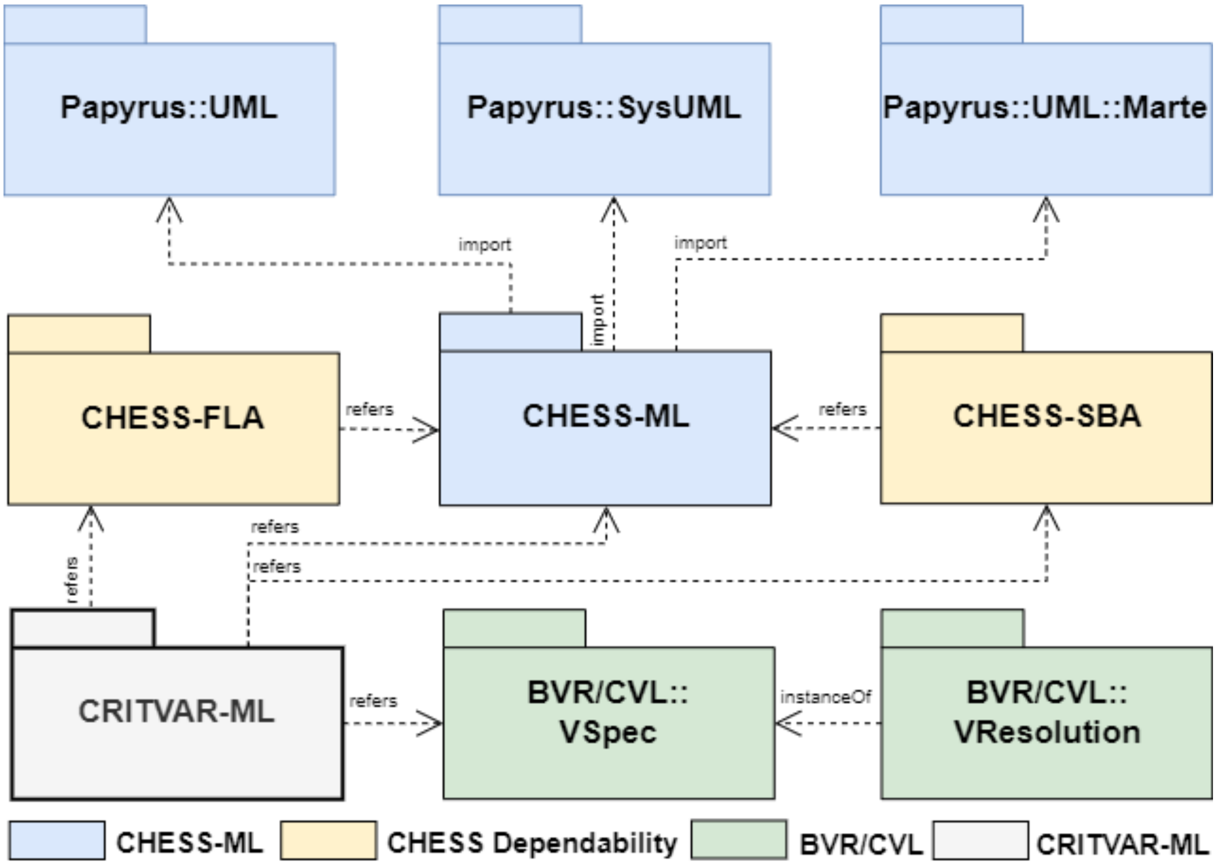


Figure 23 – Variant management into CHESS-ML models enriched with CHESS-FLA failure annotations using CRITVAR-ML.

*CRITVAR-ML::CriticalObjectExistence* is materialized into a *Papyrus UML stereotype* to bind a variability abstraction (e.g., a single *feature* or a feature expression), specified in a feature model (e.g., BVR VSpec and pure::variants vdm model), to its realization into CHESS-ML (UML/SysML) model elements (structural variability), e.g., *components* (SysML::Part), *ports* (SysML::Port), and *connections* (SysML::Connection). The following figure illustrates the use of CritialObjectExistence in a CHESS model. As shown in the figure, CRITVAR handles both fine grained and coarse grained dependencies in the model:

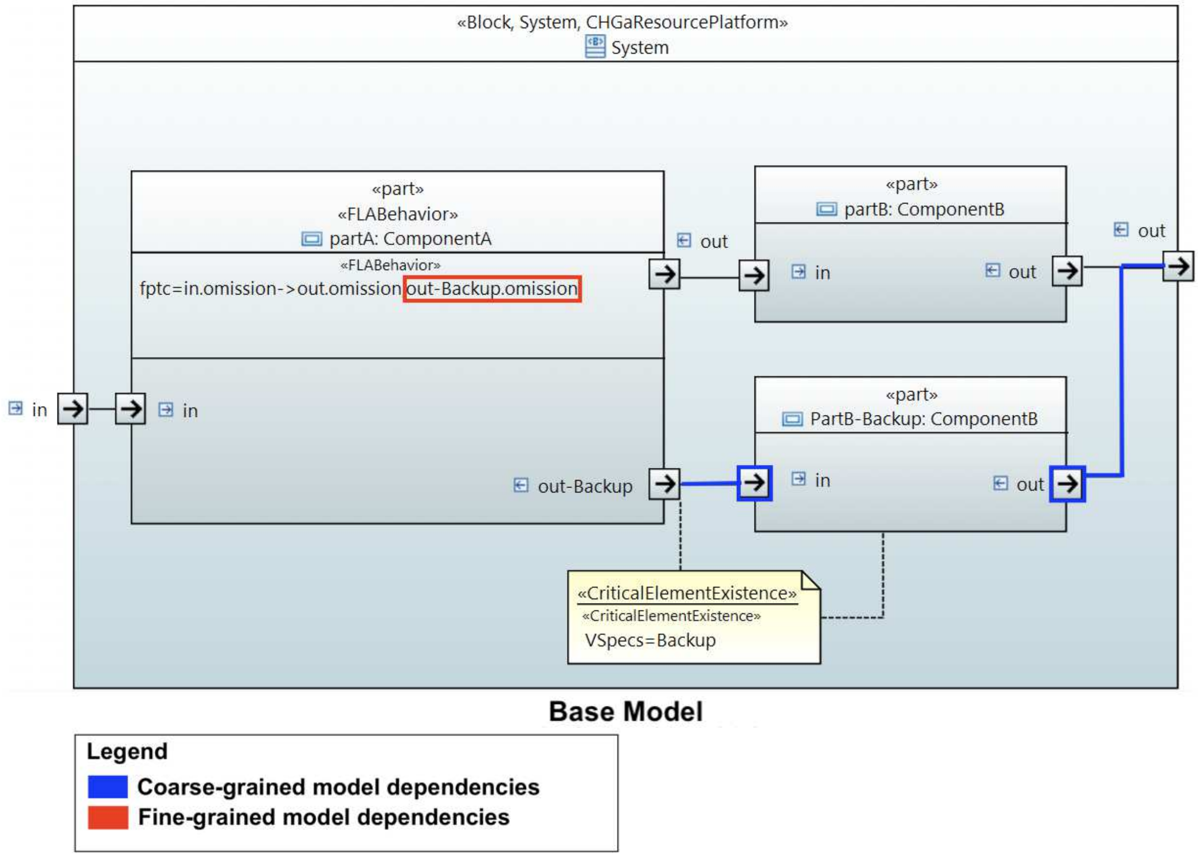

«Block, System, CHGaResourcePlatform»
System
«part»
«FLABehavior»
partA: ComponentA
«FLABehavior»
fptc=in.omission->out.omission out-Backup.omission
out
«part»
partB: ComponentB
in
out
out
in
in
out-Backup
«part»
PartB-Backup: ComponentB
in
out
«CriticalElementExistence»
«CriticalElementExistence»
VSpecs=Backup
Base Model
Legend
🟦 Coarse-grained model dependencies
🟥 Fine-grained model dependencies

Figure 24 - *CriticalObjectExistence* Variation Point, the affected coarse-grained and fine-grained model information

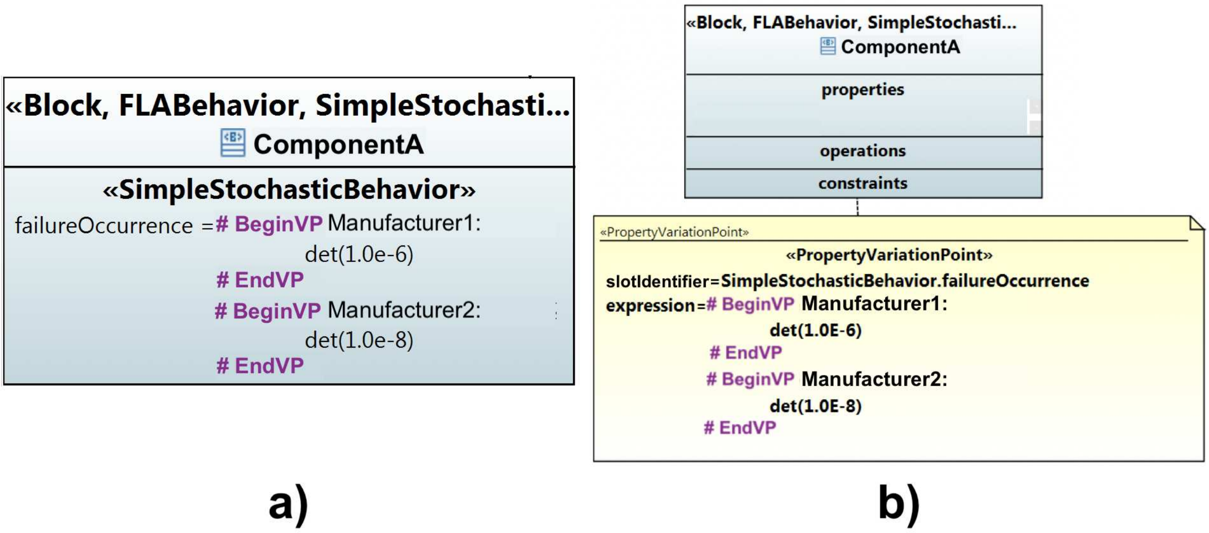
*CRITVAR-ML::DependabilityPropertyValueAssignmnet* is materialized into a *grammar* and an annotative *textual language* to bind variability abstractions (*features*) to their realization into functional safety, e.g., failure mode, and reliability, i.e., probabilistic information about component failure modes, failure and repair rate information expressed as CHESS-FLA and CHESS-SBA annotations (parametric variability).

Figure 25 illustrates an example of a variability expression derived using the production rules presented above.

```
#BeginVP Feature1 AND NOT Feature2:
    /* Kept if Feature1 is resolved as True and Feature2 is not */
     VariableValue1
    #BeginVP Feature3:
        /* Kept when Feature1 and Feature3 are
          resolved as True and Feature2 is not */
         VariableValue2
    #EndVP
#EndVP
```

Figure 25 – Fine-grained variability specified through variability expressions

In CRITVAR-ML for CHESS there are two equivalent ways to specify fine-grained variability into model artifacts. One way is by enclosing varying dependability information directly with **#BeginVP** and **#EndVP** tags as shown in Figure 25 a). If the VSpec *Manufacturer1* is resolved as true, then *ComponentA* will have a *failureOccurrence* of det(1.0e-6). If the VSpec *Manufacturer2* is resolved as true instead, then *ComponentA* will have a *failureOccurrence* of det(1.0e-6). Another way to manage fine-grained dependability on dependability information is by annotating properties with DependabilityPropertyValueAssignment comments. Figure 25 b) shows a component annotated with a DependabilityPropertyValueAssignment comment. The varying information is within the *failureOccurrence* slot of the *SimpleStochasticBehavior* stereotype applied to *ComponentA*.


«Block, FLABehavior, SimpleStochasti...
ComponentA
«SimpleStochasticBehavior»
failureOccurrence =# BeginVP Manufacturer1:
det(1.0e-6)
# EndVP
# BeginVP Manufacturer2:
det(1.0e-8)
# EndVP

«Block, FLABehavior, SimpleStochasti...
ComponentA
properties
operations
constraints

«PropertyVariationPoint»
«PropertyVariationPoint»
slotIdentifier=SimpleStochasticBehavior.failureOccurrence
expression=# BeginVP Manufacturer1:
det(1.0E-6)
# EndVP
# BeginVP Manufacturer2:
det(1.0E-8)
# EndVP

Figure 26 - Application of the DependabilityPropertyValueAssignment variation point to manage fine-grained dependability information

## 4.4 SUMMARY

This section has described the CRITVAR Modeling Language and its constructs to management variability at higher and lower levels of granularity. In the next chapter, we discuss the CRITVAR Model Transformation Engine to support the automatic derivation of configured models, based on the mappings created using the concepts presented in this chapter.

# 5 CRITVAR MODEL TRANSFORMATION ENGINE - TOOLING

This chapter introduces CRITVAR Model Transformation Engine (CRTIVAR-MTE)[1] to support the resolution of variability into CHESS-ML models enriched with functional safety and security analysis information. Thus, answering **RQ2**: How to derive correct and complete (with respect to feature selection) product system models enriched with dependability information from a variability model? CRITVAR-ML variation points was implemented by extending Papyrus UML (ECLIPSE FOUNDATION, 2017) with CRITVAR-ML variability profile. Papyrus was chosen for being an open-source and mature modeling environment. Papyrus UML supports the specification of UML 2.5 (OMG, 2017d) and SysML 1.1 (OMG, 2017b) modes, and many other UML and SysML profiles, e.g., CHESS (MAZZINI et al., 2016), UML MARTE (OMG, 2019b), and AUTOSAR (AUTOSAR, 2006) automotive industry standard for model-based design, and dependability modeling profiles (GALLINA; JAVED, et al., 2012; MONTECCHI; PURI, 2020).

CRITVAR-MTE supports the resolution of variability into CHESS-ML models enriched with dependability information for the moment, but it could be extended to be compatible with other UML/SysML modeling environments, e.g., Eclipse Capella [2]. Since UML and UML profiles are based on XMI standardized format, the portability between different solutions should be straightforward. Section 5.1 provides an overview of CRITVAR-MTE architecture. 5.2 describes the variability resolution algorithms. Finally, 5.3 presents a summary of this chapter.

## 5.1 CRITVAR Model Transformation Engine Architecture

The CRITVAR model transformation engine supports the automatic derivation of a configuration model based on the feature selection expressed in an instance of the product family feature model. The transformation engine was implemented in Python using the ElementTree (ORG., 2020) library for querying XMI and UML models. PyParsing (PYPARSING, 2020) module was used for parsing element property variability expressions.

Figure 27 provides an overview of CRITVAR Model Transformation Engine (CRITVAR-MTE) architecture. CRITVAR-MTE requires the following input artefacts: a base model from the targeting Model Editor, e.g., CHESS-ML (Papyrus) with CRITVAR-ML variability annotations, one product family feature model, and one or a set of product configuration feature models produced by a variability modeling tool, e.g., BVR (HAUGEN; ØGÅRD, 2014), pure::variants (PURE-SYSTEMS, 2021).

[1] CRITVAR-MTE Download: https://github.com/bressan3/CRITVAR-MTE-Public

[2] https://www.eclipse.org/capella/

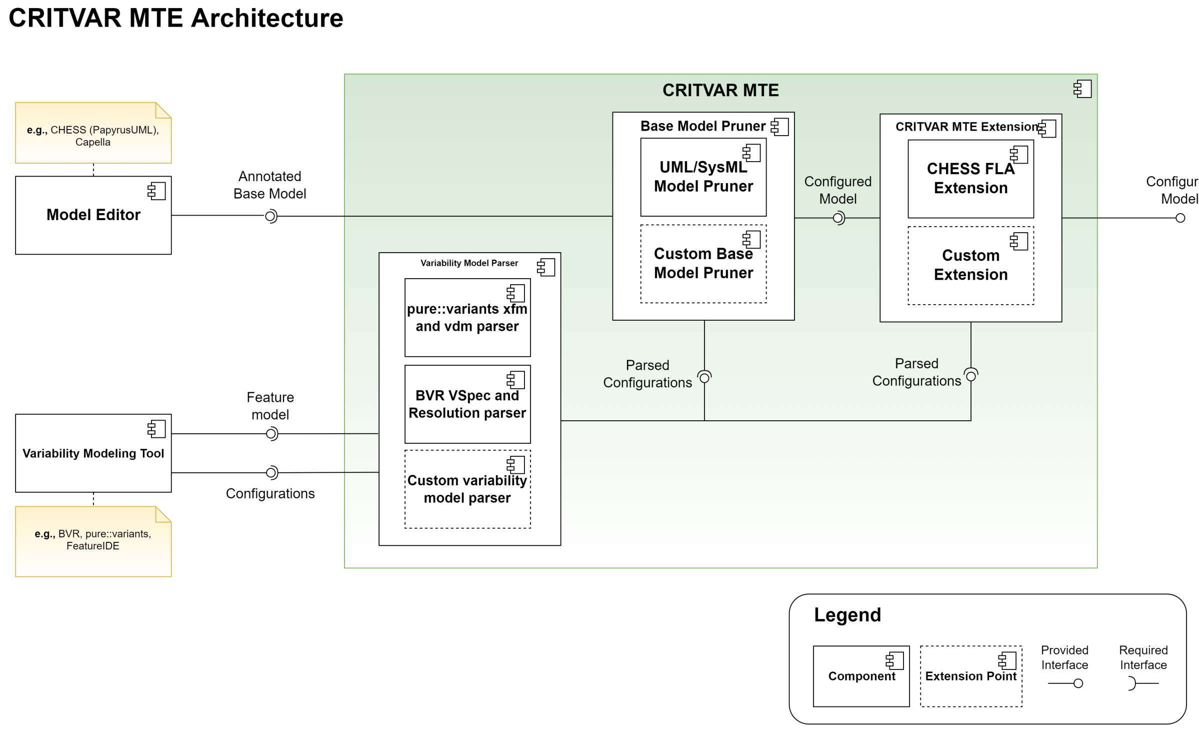


Figure 27 - CRITVAR Model Transformation Engine Component Diagram

The CRITVAR-MTE::Variability Model Parser (Figure 27) is responsible for parsing the product configuration feature models and pass the information to the *Base Model Pruner* resolving variability into the base model based on the selection features expressed in the input *Product Configuration* feature model(s). The *Base Model Pruner* is detailed in section 5.2. The current implementation of *CRITVAR-MTE::Variability Model Parser* is compatible with pure::variants and BVR variability models, i.e., product family and configuration models, and *CRITVAR-MTE::Base Model Pruner* supports the resolution of variability into *CHESS UML/SysML* base models enriched with *CHESS-FLA* dependability annotations via *CRITVAR-MTE::CHESS FLA Extension*. CHESS-FLA Extension supports variability resolution (pruning) into fptc (Failure Propagation Transformation Calculus) rules based on the interfaces (i.e., CHESS-ML component input and output ports and their connections) present into the configured (product) model.

The CRITVAR Model Transformation Engine can be extended to enable compatibility with other variability modeling solutions (Custom variability model parser component), e.g., FeatureIDE (MEINICKE et al., 2017), and base models (*Custom Base Model Pruner* component), e.g., MATLAB Simulink (MATHWORKS, 2021) models enriched with HiP-HOPS (PAPADOPOULOS et al., 2011) failure annotations extensions (*Custom Extension*). CRITVAR-MTE provides standardized interfaces (see Figure 28), e.g., via abstract classes, which can be extended with concrete implementations to enable compatibility with third part variability models (*VariabilityModel* class), base models (*BaseModelParser* class), and base model extensions (*CRITVARExtensions* class).

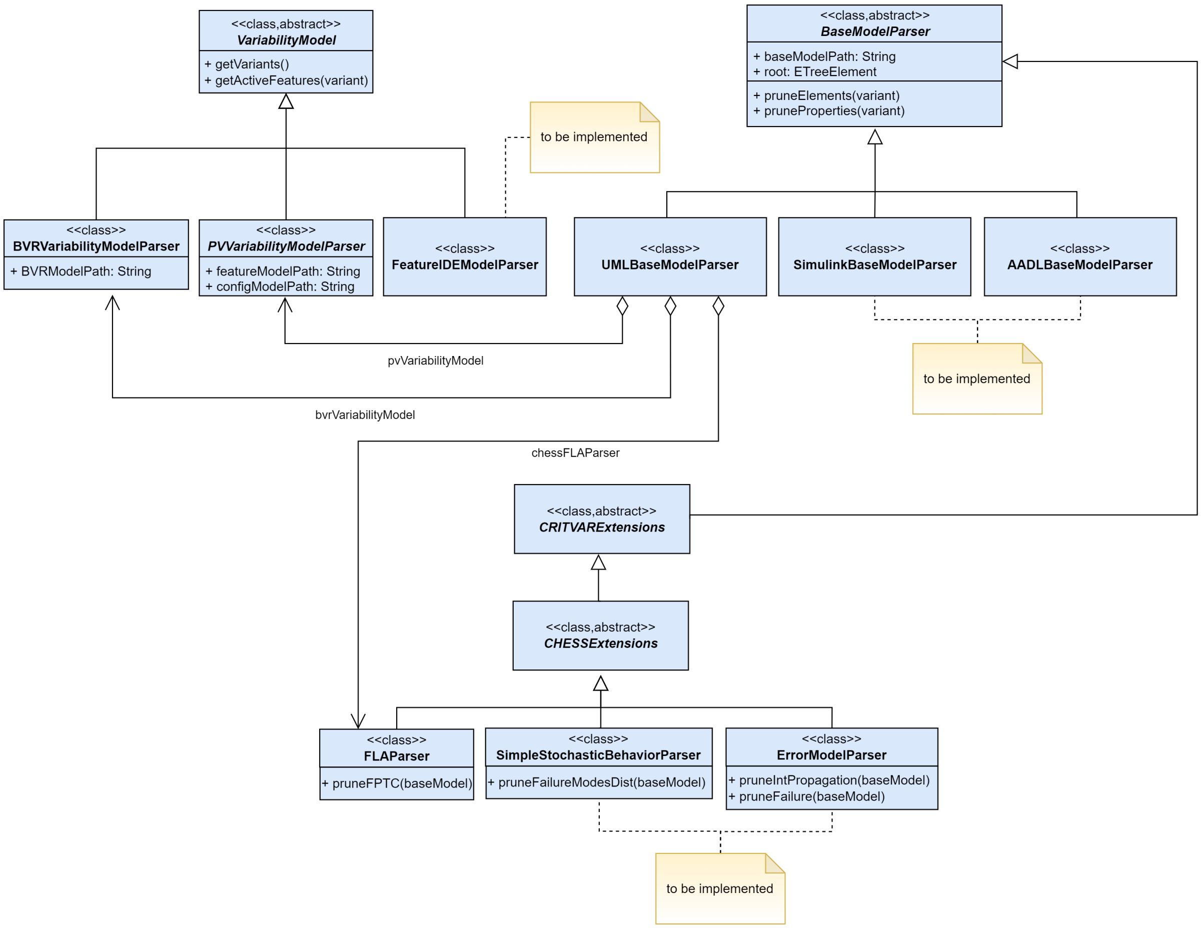


Figure 28 - CRITVAR Model Transformation Engine Class Diagram

*CRITVAR-MTE::VariabilityModel* class provides *getVariants* and *getActiveFeatures* abstract methods, which should be implemented by concrete classes to support the parsing of third part variability models. The current implementation of CRITVAR-MTE provides support for parsing BVR *VSpec* models, and pure::variants feature (*.xfm*) and variant description (*.vdm*) models. Other extensions of *VariabilityModel* class can be implemented to support the parsing of other third part variability models, e.g., FeatureIDE (MEINICKE et al., 2017). The support for parsing FeatureIDE feature model can be implemented in future releases of CRITVAR-MTE.

*CRITVAR-MTE::BaseModelParser* has *baseModelPath* and *root* attributes, and it provides *pruneElements(variant)* and *pruneProperties(variant)* abstract methods to be implemented by concrete classes to enable variability resolution (pruning) into the targeting base model in conformance with feature selection expressed into a variant feature model (e.g., *VSpec*, *.vdm* model). *baseModelPath* represents the base model file path, and *root* is a nested composite data structure to manipulate the input base model file. A base model is a model that contains core, optional, and alternative elements, which can be combined to derive two or more different configurations (variants). A *UML Class Diagram*, *SysML Block Definition* and *Internal Block* diagrams with core and alternative elements are examples of base models. Concrete implementations of *BaseModelParser.pruneElements(variant)* support the resolution of variability into base model elements, e.g., UML::Class, UML::Association. Implementations of *BaseModelParser.prunePro-perties(variant)* enable variability resolution into model element property and their values, e.g., property name from an UML::Class or UML::Port and its value, which may change from a system variant to another. UMLBaseModelParser is a concrete implementation of *BaseModelParser* that supports the resolution of variability into CHESS (MAZZINI et al., 2016) and Papyrus (ECLIPSE FOUNDATION, 2017) UML and SysML standard diagrams.

*CRITVAR-MTE::CRITVARExtensions* is an abstract class that should be implemented to enable the resolution of variability into model elements, properties, property values specified using specific extensions from the core base model language, e.g., CHESS-FLA failure logic and stochastic annotations into elements from a CHESS ML Block Definition Diagram. *CRITVAR-MTE::CRITVARExtensions* extends *BaseModelParser* where implementations for *pruneElements(variant)* and *pruneProperties(variant)* should be provided by concrete classes, to enable variability resolution into elements and properties specified using an extension (e.g., CHESS-FLA) from the core base model language (e.g., CHESS ML).

CRITVAR-MTE::FLAParser is a concrete implementation of *CRIVARExtensions* abstract class that supports variability resolution into fragments of CHESS-FLA failure logic annotations (Failure Propagation Transformation Calculus rules) attached to CHESS ML components and linked to component ports, e.g., *in.omission -> out.omission, out-Backup.omission* rule linked to an instance (*partA*) of ComponentA (see Figure 7) and its ports. The *out-Backup* linked to an partB component instance input port is an optional element that could be present or absent in a product configuration according to feature selection.In the example from Figure 7, when *Backup* feature is not selected, *partA.out-Backup* output port and its connections, and the *out-Backup.omission* fragment from the FTPC rule are removed from model after variability resolution and product derivation. CRTIVAR-MTE::FLAParser supports parsing and pruning of CHESS ML models annotated with FPTC rules based on feature selection expressed in the product feature model. Concrete implementations of *CRITVARExtensions* abstract class to support the resolution of variability into stochastic (SimpleStochasticBehaviorParser class) annotations and elements from error model state machines (ErrorModelParser class) attached to CHESS ML components will be provided in future releases of CRITVAR-MTE. The current implementation of CRITVAR-MTE supports variability resolution into CHESS ML and Papyrus UML models enriched with dependability (functional safety and security) information.

CRITVAR-MTE::BaseModelParser and CRITVARExtensions can also be extended to enable the resolution of variability into base models and extensions from different languages provided by different vendors. In future, we intend to extend CRITVAR-MTE to enable variability resolution into MATLAB/Simulink (MATHWORKS, 2021) models enriched with HiP-HOPS (PAPADOPOULOS et al., 2011) dependability annotations, ANSYS Medini (ANSYS, 2020a) system model enriched with safety and cyber-security information, and OSATE AADL (DELANGE, 2016) models and their Error Annex (FEILER, 2013) error state machine diagrams.

## 5.2 CRITVAR Model Transformation Engine Algorithms

This section describes the concrete implementations for *CRITVAR-MTE::BaseModelParser prune elements* and *prune properties* (see Figure 28) methods. *Prune elements* algorithm supports the resolution of variability into UML/SysML base model elements annotated with CRITVAR-ML «ElementVariationPoint» stereotype. *Prune properties* enables the resolution of variability at model element property values surrounded by #BeginVP #EndVP annotations.

### **5.2.1** Model Element Transformation Algorithm

The pruning elements algorithm, shown in Listing 1, resolves variability in CHESS ML (UML and SysML) base model elements annotated with «CriticalElementExistence» variation point stereotypes.

Model elements mapped by CriticalElementExistence are resolved according to function pruneModelElements described in 1. The function receives the base model (model) and the its configurations (configs) as inputs.

The configs parameter comprises of a dictionary of lists where each entry is a configuration. For each entry representing a configuration in the dictionary, there is a list containing the names of the features that have been activated for that particular configuration.

Initially, the algorithm iterates through each entry representing a configuration in the configs dictionary (line 2). Then, the algorithm assigns the model contents to the variable prunnedModel (line 3). For each config, the algorithm iterates through each elementVariationPoint in the prunnedModel (line 4).

A criticalElementExistence contains a feature expression and annotated model elements. If the feature expression in the current criticalElementExistence is true when compared to the features in the current config (line 5), e.g., criticalElementExistence.Features = “Feature1 AND NOT Feature2” and config = [Feature1], then the current criticalElementExistence shall stay in the config and therefore the algorithm iterates into the next criticalElementExistence (line 6). Else (line 7), the current criticalElementExistence is removed from the model alongside all its annotated model elements and dependencies of those elements (line 8).

Once the algorithm is done iterating through all criticalElementExistence and transforming the model according to the current config, the algorithm writes the resulting prunnedModel to a file (line 11). Then the algorithm repeats the same set of steps once again for the next config. A new file containing the configurations, is created for each config.

**Algorithm 1:** Pruning of model elements annotated with CriticalElementExistence variation point comments. **Inputs**: base *model*, configuration feature models *configs*.

```
1  Function pruneModelElements(model, configs):
2      foreach config ∈ configs do
3          prunnedModel = model;
4          foreach criticalElementExistence ∈ prunnedModel do
5              if checkExpr(config, criticalElementExistence.Features) then
6                  next();
7              else
8                  removeWithDeps(criticalElementExistence);
9              end
10         end
11     end
12     return;
13
```

### 5.2.2 Element Property Transformation Algorithm

Model elements mapped by DependabilityPropertyValueAssignment are resolved according to functions pruneModelPropertiesDirect() and pruneModelPropertiesComment() described in Algorithms Y and Z. pruneModelPropertiesDirect() is responsible for pruning model properties annotated directly as show in Figure 9 a). pruneModelPropertiesComment() is responsible to address properties annotated via the DependabilityPropertyAssignment stereotype as depicted in Figure 9 b).

Both Algorithms and receive the base model (model) and the its configurations (configs) as inputs. A detailed description of these input parameters is given in the previous section.

The first action for both algorithms, is to iterate though each config provided as input to their respective functions. Then, the algorithm assigns the model contents to the variable prunnedModel.

In , the function iterates through each DependabilityPropertyValueAssignment comments in the model (line 4), i.e., propertyVariationPoint attribute in the algorithm. Then, we call the pruneExpression() function (line 5) to prune the expression within the propertyVariationPoint, (based on the features included in the current config). The pruning occurs based on the rules shown in Figure 8.

At last, the function writes the appropriate property of all elements annotated by the current propertyVariationPoint, with the value of prunedExpression, via annotateProperty() (line 6). Once done iterating through all propertyVariationPoints, the function writes the transformed model into a file (line 8). The same process is repeated for the remaining configs.

**Algorithm 2:** Pruning of model elements annotated with DependabilityPropertyAssignment comments

```
1  Function pruneModelProperties(model, configs):
2      foreach config ∈ configs do
3          prunnedModel = model;
4          foreach propertyVariationPoint ∈ prunnedModel do
5              prunedExpression = pruneExpression(config,
                 propertyVariationPoint.expression);
6              annotateProperty(prunedExpression,
                 propertyVariationPoint.annotatedElements);
7          end
8          write(prunnedModel, config + ".uml");
9      end
10
```

In Algorithm 3, the function iterates through each element in the model (line 4). For each property in the element, it parses it and verifies if it complies with the grammar illustrated in Section 3.5.2 (line 6). If so, then we assign the property with the value returned by pruneExpression() (line 7). Else, the algorithm iterates into the next property. As in the other algorithms, once done iterating through all elements, the function writes the transformed model into a file (line 8). The same process is repeated for the remaining configs.

**Algorithm 3:** Pruning of model properties annotated directly

```
Function pruneModelProperties(model, configs):
    foreach config ∈ configs do
        prunnedModel = model;
        foreach element ∈ prunnedModel do
            foreach property ∈ element.properties do
                if isEnrichedWithVariability(property.value) then
                    property.value = pruneExpression(property.value);
                else
                    next();
                end
            end
        end
        write(prunnedModel, config + ".uml");
    end
    return;

```

## 5.3 DISCUSSION

This section has described the CRITVAR Model Transformation Engine. The CRITVAR Model Transformation Engine currently works with UML/SysML base models annotated with the CRITVAR stereotypes. Extension points can be written to accommodate other types of system, dependability and variability modeling tools.

Section 5.1 provided an overview of the CRITVAR MTE architecture and section 5.2 provided the descriptions of the inner workings of the Element and Element Property variability transformation algorithms provided as part of the CRITVAR MTE.

In the following section we present the CRITVAR Process which combines traditional product lines and safety/security analysis processes together to support the dependability analysis of dependable systems.

# 6 THE CRITVAR SYSTEMATIC PROCESS - METHOD

With the objective of answering **RQ3**: How model-based safety and cybersecurity co-analyses can be integrated within product line processes?, this chapter introduces the CRITVAR method, which is a process to support variant management and the systematic reuse of safety and security model artifacts. The CRITVAR Method is built upon the DEPendable-SPLE (OLIVEIRA, Andre Luiz de; BRAGA, R. T. V.; MASIERO, P. C.; PAPADOPOULOS, et al., 2018) methodology and it provides a systematic process that introduces dependability engineering and analysis activities within traditional Software Product Line Domain and Application engineering phases. The process supports variant management and the systematic reuse of SysML models enriched with dependability information for reliability and safety analysis, and threat analysis and risk assessment as illustrated in Figure 29. The CRITVAR Method relies on the integration between CHESS, Papyrus UML, and CRITVAR, and it is based on the activities prescribed by the ISO 26262 (ISO, 2018) safety and ISO/SAE 21434 (ISO, 2021) cybersecurity standards.

The Section provides an overview of CRITVAR Process main Phases (Domain Engineering and Application Engineering) to support variant management and the systematic reuse of system models for dependability analysis. Section **6.1.1** details the Domain Engineering Phase, with its three main activities: Domain Analysis, Domain Design and Implementation, and Domain Management. 6.2 presents the Application Engineering Phase, and its three main activities: Product Requirements Engineering, Product Design and Implementation, and Product Analysis. Finally, Section 5.3 provides the discussion of this chapter.

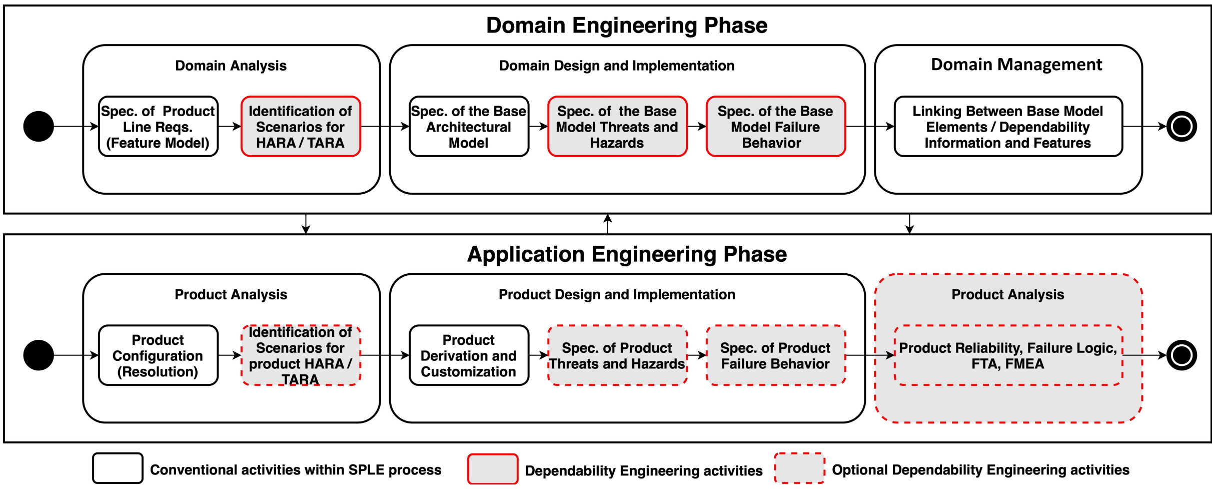

Domain Engineering Phase
Domain Analysis
Spec. of Product Line Reqs. (Feature Model)
Identification of Scenarios for HARA / TARA
Domain Design and Implementation
Spec. of the Base Architectural Model
Spec. of the Base Model Threats and Hazards
Spec. of the Base Model Failure Behavior
Domain Management
Linking Between Base Model Elements / Dependability Information and Features
Application Engineering Phase
Product Analysis
Product Configuration (Resolution)
Identification of Scenarios for product HARA / TARA
Product Design and Implementation
Product Derivation and Customization
Spec. of Product Threats and Hazards
Spec. of Product Failure Behavior
Product Analysis
Product Reliability, Failure Logic, FTA, FMEA
Conventional activities within SPLE process
Dependability Engineering activities
Optional Dependability Engineering activities

Figure 29 - The CRITVAR Systematic Process

## 6.1 CRITVAR PROCESS: DOMAIN ENGINEERING

This phase encompasses three main sub-phases: Domain Analysis, Domain Design and Implementation, and Product Management.

### 6.1.1 Domain Analysis

- **Inputs:** Domain knowledge and requirements;
- **Output:** Feature model;
- **Description:** This is the starting point of the Domain Engineering phase covering the specification of product line features, their relationships, and constraints using a feature model editor e.g., BVR VSpec or pure::variants. The feature model contains system and usage context features i.e., characteristics of the operating environment, how and where a system feature can be used (KANG, K. C. et al., 1998). The feature model provides higher-level abstractions to manage and trace variation across different product variants.

The Domain Analysis comprises two activities: Specification of Product Line Requirements (Feature Model) and Identification of Scenarios for HARA/TARA.

#### *6.1.1.1 Specification of Product Line Requirements*

- **Inputs:** Domain knowledge and requirements;
- **Outputs:** Feature model;
- **Description:** Specification of product line features, their relationships, and constraints using a feature model.

#### *6.1.1.2 Identification of Scenarios for HARA/TARA*

- **Inputs:** Feature model.
- **Outputs:** Scenarios for HARA/TARA and refined feature model.
- **Description:** Identify the different scenarios for HARA and TARA that can happen across the different configurations. Refine the feature model if necessary.

### 6.1.2 Domain Design and Implementation

- **Inputs:** Feature model;
- **Outputs:** base model containing architectural and dependability information.

- **Description:** The system and context variation points and their variants, relationships, and constraints, stored in the feature model, drive product line design. The realization of system features in the design is specified as blocks, parts, flow ports, connectors, and properties when using SysML. The realization of system and usage context features and their relationships into dependability information is specified as annotations in the elements of the design. For example, when using the CHESS dependability analysis tool, security annotations, e.g., threats and failure logic e.g., FLABehavior and SimpleStochasticBehavior model annotations, can be attached to design components using the CHESS-ML stereotypes.

This sub-phase has three main activities: Specification of Base Architectural Model, Specification of the Base Model Hazards and Threats, and Specification of the Base Model Failure Behaviour.

#### *6.1.2.1 Specification of Base Architectural Model*

- **Inputs:** High-level description of the system including functionality and interfaces to other systems. Feature model resulting from the previous step.
- **Outputs:** Architectural model of the system.
- **Description:** In this activity, we specify the architectural model of the system including components, interfaces, information flow, and behavior.

#### *6.1.2.2 Specification of Base Model Threats and Hazards*

- **Inputs:** Architectural model of the system.
- **Outputs:** Hazards and Threats.
- **Description:** In this step, we identify the hazards and threats that apply to the system based on the architectural model defined in the previous step.

#### *6.1.2.3 Specification of Base Model Failure and Security Threat Behavior*

- **Inputs:** Architectural model of the system, hazards, and cybersecurity threats.
- **Outputs:** Component dependability information including failure behavior and attack paths.
- **Description:** In this step, we identify how individual components in the architecture contribute to system-level threats and hazards.

### 6.1.3 Domain Management

The Domain Management sub-phase has one activity, the specification of mappings between Base Model Elements/Dependability Information and features.

- **Inputs:** Feature and base models containing architectural and dependability information;
- **Outputs:** Base model enriched with CRITVAR variation points i.e., variability information, linking features to model elements and dependability information;
- **Description:** During the Domain Management sub-phase, links between domain features and their realization into architectural and dependability models are established using CRITVAR *ElementVariationPoints* and *PropertyVariationPoints*. This activity covers the definition of variation points containing elements and information that vary across product variants and their links to domain features.

## 6.2 CRITVAR PROCESS: APPLICATION ENGINEERING

This Phase includes the same sub-phases performed throughout the Domain Engineering phase from the product perspective. It encompasses three sub-phases: Product Requirements Engineering, Product Design and Implementation, and Product Analysis.

### 6.2.1 Product Analysis

- **Inputs:** product requirements and the domain feature model;
- **Outputs:** Resolution models containing the product configurations and their implemented features;
- **Description:** In this sub-phase the application engineer configures the resolution model by selecting the domain features that address product requirements. If required, the application engineer may also identify newer product-specific HARA/TARA scenarios.

This sub-phase has two activities: Product Configuration/Resolution and Identification of Scenarios for Product HARA/TARA.

#### *6.2.1.1* Product Configuration/Resolution

- **Inputs:** product requirements and the domain feature model;
- **Outputs:** Resolution models containing the product configurations and their implemented features;

- **Description:** In this activity the application engineer configures the resolution model by selecting the domain features that address product requirements.

#### *6.2.1.2* Identification of Scenarios for Product HARA/TARA

- **Inputs:** Product configurations.
- **Outputs:** Product-specific scenarios for HARA/TARA.
- **Description:** Identify additional HARA/TARA scenarios for an individual product variant.

### 6.2.2 Product Design and Implementation

- **Inputs:** base model containing architectural, dependability, and variability information. Resolution models;
- **Outputs:** product models containing the selected architectural elements and dependability information
- **Description:** Once the products are fully configured, the resolution and base models are fed into the CRITVAR Model Transformation Engine, and the product-specific models are generated. The models are produced according to the features selected in the resolution model. Once a new product is derived, new architectural elements or dependability information can be added to the model if needed. If necessary, the application engineer may also extend the model with additional product-specific threats, hazards, and component failure behaviors.

This sub-phase comprises the following activities: Product Derivation and Customization, Specification of Product Safety Hazards and Security Threats, and Specification of Product Failure Behaviour.

#### *6.2.2.1* Product Derivation and Customization

- **Inputs:** base model containing architectural, dependability and variability information. Resolution models;
- **Outputs:** product models containing the selected architectural elements and dependability information
- **Description:** Once the products have been fully configured, the resolution and base models are fed into the CRITVAR Model Transformation Engine, and the product-specific models are generated. The models are produced according to the features selected in the resolution model.

#### *6.2.2.2 Specification of Product Safety Hazards and Security Threats*

- **Inputs:** Architectural model of the configured system.
- **Outputs:** Additional product-specific Threats and Hazards.
- **Description:** In this step, we identify additional hazards and security threats that apply to the configured system based on the configured architectural model defined.

#### *6.2.2.3* Specification of Product Failure Behavior

- **Inputs:** Architectural model of the configured system, threats and hazards.
- **Outputs:** Additional product-specific component dependability information including failure behavior and attack paths.
- **Description:** In this step, we identify additional information on how individual components in the architecture contribute to system-level threats and hazards of configured products.

### 6.2.3 Product Analysis

This sub-phase comprises one activity: Product Reliability, Failure Logic, FTA, and FMEA synthesis.

#### *6.2.3.1* Product Reliability, Failure Logic, FTA, and FMEA Synthesis

- **Inputs:** product models;
- **Outputs:** product reliability, failure logic, Fault Tree, FMEA, and threat analysis and risk assessment (TARA) results;
- **Description:** The product models derived by the CRITVAR Model Transformation Engine, can be fed into the CHESS-FLA, CHESS-SBA plugins for obtaining failure logic and state-based analysis.

## 6.3 SUMMARY

This chapter described the CRITVAR method. The CRITVAR process is built upon the traditional product line engineering processes, introducing additional dependability engineering activities in the flow.

# 7 EVALUATION

In this chapter, we evaluate the proposed CRITVAR methodology and model transformation engine on a Highly Automated Driving (HAD) vehicle system from the automotive domain. The evaluation was performed through a comparative study in which we compared the application of our methodology with similar variability modeling and management solutions (i.e., pure::variants and BVR). The evaluation is organized in the following sections: planning, design, preparation of the execution workflow, execution, data collection, and threats to validity.

## 7.1 PLANNING

We considered the reference architecture of a variant-intensive Highly Automated Driving (HAD) Vehicle (MUNK; NORDMANN, 2020) to address Advanced driver-assistance systems use-cases. The reason for choosing the HAD in this evaluation is because ADAS systems typically involve both safety and security concerns.

The HAD Vehicle may include up to four cameras, an automated powertrain, and steering systems. Its cameras and automated systems are both connected to a central vehicle computer which is reused across all vehicle variants. The vehicle computer sends commands to the powertrain and steering systems when automatic cruise control and automatic steering are implemented. These commands are calculated based on the information captured by the vehicle cameras and used to control its longitudinal and lateral movements.

For the case study, we chose the OpenCert CHESS Client - AMASS Tool Platform Prototype[1]. AMASS comprises an European wide open tool platform, to address the design and the dependability analysis of contemporary critical systems (VARA et al., 2019). The platform is a result of a joint effort involving certification authorities, component suppliers, manufacturers, research institutions, and tool vendors. It serves as an extension of previous successful EU projects, and it integrates state-of-the-art model-based design (i.e. Papyrus[2]), variability management (i.e. BVR (VASILEVSKIY et al., 2015)), failure modeling (i.e. CHESS (MAZZINI et al., 2016)) and dependability analysis solutions (i.e. CHESS Failure Logic (**Gallina2012AArchitectures**) and CHESS State-Based Analyses [3] (MONTECCHI; GALLINA, 2017)).

The first variability modeling and management solution considered in our study is BVR, a language (HAUGEN, 2014a) and open source tool bundle to support variability management activities (e.g. feature modeling, mapping between features and external

[1] https://www.eclipse.org/opencert/downloads/

[2] https://www.eclipse.org/papyrus/

[3] https://github.com/montex/CHESS-SBA

model fragments and product resolution/configuration) (VASILEVSKIY et al., 2015). The main reason for choosing BVR for our comparative study, lies in the fact that it is open-source and already comes bundled with a mature model-based design and dependability analysis solution (i.e. AMASS Tool Platform). Furthermore, BVR has also been previously applied in different occasions to manage variability within safety process (JAVED; GALLINA, 2018) and safety-critical models (JAVED; GALLINA; CARLSSON, 2019). The BVR Tool Bundle comprises a set of independent and standalone editors. The VSpec model editor supports the definition of VSpec (feature) models according to the BVR language. BVR extends traditional feature modeling with new concepts such as multiplicity, constraints, and variables. The Realization model editor provides an interface for mapping domain features into external model fragments through fragment substitutions containing placements and replacements. Fragment substitutions when activated by a feature, remove the elements within its placements and replace them with model fragments in its replacements. The resolution model editor supports the instantiation and configuration of feature models by setting features as true or false. The resolution model is used by the BVR model transformation engine for automatically transforming the base model according to the selected features.

Since CRITVAR only provides a modeling language to support the mapping between model artifacts and features and a model transformation engine, the BVR VSpec and Resolution model editors were used alongside it, to support feature modeling and to provide the CRITVAR Model Transformation Engine with component configurations.

The second variability management solution considered was pure::variants and its plugin connector for EMF (Eclipse Modeling Framework) models. Similar to CRITVAR, pure::variants also works with the idea of 150% variability in which model fragments, are either activated or deleted from the final configured models, according to feature selection. Pure::variants provides a Feature Model editor in which features, variables, constraints and multiplicities can be specified. The Mappings Editor provides an interface for linking domain features (or feature expressions) with model element fragments (using "conditions") and property values (using "calculations"). At last, the Variants Editor enables the configuration of new model variants through the selection of features from the domain feature model. Variant models are further used by the pure::variants Model Transformation Engine to generate configured 100% models.

## 7.2 STUDY DESIGN

We structured the evaluation using the Goals-Questions-Metrics (GQM) approach (BASILI; CALDIERA; ROMBACH, 1994):

> **Analyze** CRITVAR, BVR and pure::variants **for the purpose of** evaluating software variability management techniques **with respect to** their ability to reduce the gap between variability constructs and dependability artefacts, and ensuring the derivation of correct system models enriched with dependability information **from the point of view of the** product line engineer **in the context of** variant-intensive critical systems.

Table 1 illustrates the goals, questions, and metrics that guided our evaluation aiming at answering the **RQ4** "*Which software variant management technique is most effective and efficient in supporting product line engineers in managing the diversity into safety and security artifacts throughout variability realization?*", raised in the introduction.

.

Table 1 – Evaluation Goals, Questions and Metrics

**G1:** Reduce the gap between variability constructs and dependability models.
- **Q1:** How much effort is needed for expressing variability in dependability in the domain variability model?
  - **M1:** Number of variability constructs (i.e., features and constraints) in the feature model.
- **Q2:** How much effort is needed to map variability constructs to dependability artifacts?
  - **M2:** Number element mappings.
  - **M3:** Number of dependability annotation mappings.

**G2:** Ensuring the derivation of correct configuration models with dependability information.
- **Q3:** How correct is the derived product configuration model with respect to feature selection?
  - **M4:** Variant models are complete (Boolean).

**M1** captures the number of feature model elements concerns features, constraints and variables. **M2** considers the number of elements used for mapping features to model elements. In BVR these mapping are expressed via placements, replacements, fragment substitutions. In pure::variants, the mappings are done through conditions. In CRITVAR, mappings to model elements are done using *ElementVariationPoints*. **M3** considers the number of mappings used to map variability to fine grained dependability information. In pure::variants, such mapping is done using Calculations. In CRITVAR, dependability annotation mappings are expressed via the *#BeginVP #EndVP* tags. *M4* is a boolean variable that is TRUE if the variant models are complete and FALSE case the contrary.

## 7.3 EXECUTION WORKFLOW

The factor under analysis in our evaluation is variability management techniques in the context of dependability artifacts. We considered three treatments i.e., CRITVAR, BVR and pure::variants, for managing variability in the design and functional safety of the HAD vehicle described in Section 7.1. Although the chosen variability management techniques differ from each other in a few aspects, we considered the same HAD vehicle SysML system model for all three treatments throughout the evaluation process.

Due to the aforementioned limitations of the BVR Tool Bundle regarding the lack of support for mapping variability constructs (BVR VSpecs) to safety annotations expressed as model element properties, we considered a base model without dependability annotations when using BVR. Thus, system components had to be enriched with their corresponding dependability annotations after product derivation.

It is important to highlight however that it is theoretically possible to create base models enriched with dependability information and derive complete annotated models from them, when applying BVR. Doing so however, requires us to replicate elements, according to each different dependability annotation they may present, and use the BVR Realization editor, for mapping features to their realization into model elements with variable dependability information. When applying this strategy in the HAD Vehicle model, we obtained a domain model with 98 SysML base model elements and 81 variability constructs, i.e. feature model elements such as features and constraints. The HAD vehicle SysML model used in our study comprises 37 base model elements. Thus, we did not consider the modeling strategy of replicating model elements to mitigate the construction validity threat related to the usage of different versions of HAD vehicle SysML model with different degrees of complexity in our comparative study.

In CRITVAR and pure::variants, both architectural elements and element properties, i.e., component failure annotations, were specified in the base model. In CRITVAR, architectural elements were mapped using comments with the *ElementVariationPoint* stereotype. Moreover, the variability within dependability annotations were specified using #BeginVP and #EndVP tags inside property values.

In pure::variants, mappings between base model fragments and domain features were specified as conditions with the support of pure::variants Mappings Editor. Variability in dependability annotations was specified by creating new variables in the Feature Model and calculations in the Mapping Editor. The values of these variables were assigned using pure::variants Simple Constraint Language (pvSCL) conditionals. Such conditionals were used to guarantee that variables are assigned with their appropriate values, according to the feature selection. These values are then retrieved by calculations in the Mappings Model and propagated to the configuration model after product derivation.

The HAD vehicle SysML system model was enriched with dependability annotations using the SysML profile extensions provided by the CHESS toolset. The specification of variability in the HAD vehicle SysML model using CRITVAR, BVR, and pure::variants tools was carried out by the author of this dissertation. The metrics were collected separately by different people and had their values compared to detect any major discrepancies. All metrics were manually collected considering the HAD vehicle feature model and HAD vehicle SysML configuration model.

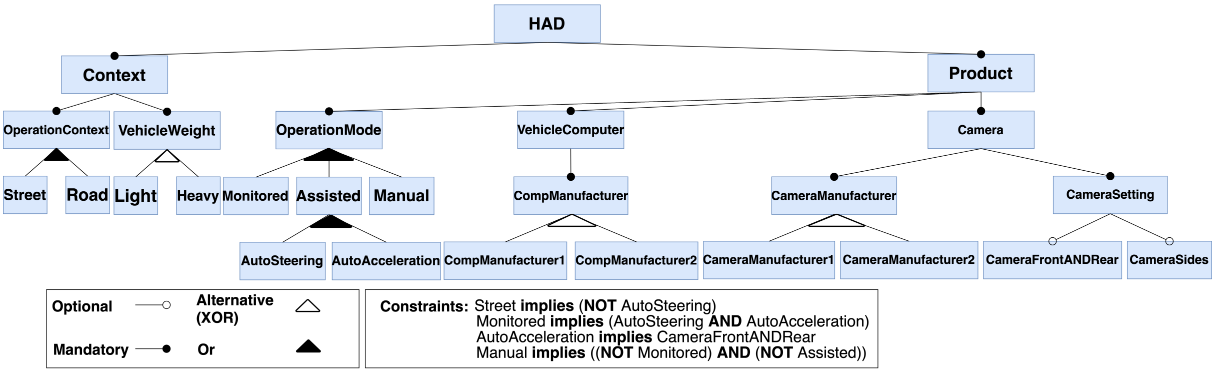
## 7.4 DATA COLLECTION

The HAD Vehicle system architecture is ought to be implemented in vehicles allowed to operate in a number of different scenarios. The HAD Vehicle model was specified following the CRITVAR Systematic Process presented in Section 7.3.

### 7.4.1 Domain Engineering Phase

#### *7.4.1.1 Domain Analysis*

The HAD Vehicle system has the following variation points: the vehicle's Weight and Operation Context, Operation Mode, Camera Settings and Manufacturer and Computer Manufacturer illustrated in the feature model from Figure 30. Changes in the feature selection may be propagated throughout HAD Vehicle design and safety annotations. The vehicles can operate under three environments: Street, Road or both. The HAD Vehicle system can be deployed in a vehicle with maximum 3.5 tons (Light) or in a truck (Heavy).



**Constraints:** Street **implies** (**NOT** AutoSteering)
Monitored **implies** (AutoSteering **AND** AutoAcceleration)
AutoAcceleration **implies** CameraFrontANDRear
Manual **implies** ((**NOT** Monitored) **AND** (**NOT** Assisted))

Figure 30 - The HAD Vehicle feature model

#### *7.4.1.2 Domain Design and Implementation and Product Management*

The HAD Vehicle can operate autonomously (Monitored feature), manually or with the assistance of the driver. In the latter, either HAD Vehicle's longitudinal or lateral movements are controlled by the driver. When in Manual mode, both steering and longitudinal movements are controlled by the driver. In this specific case, the Vehicle Computer is only responsible for alerting the driver in case the cameras detect any situation that requires further attention, e.g., the driver is too close to another vehicle. A HAD Vehicle system variant can include one of the two Vehicle Computers provided by two different manufacturers. Clients can configure their HAD Vehicle with cameras from two

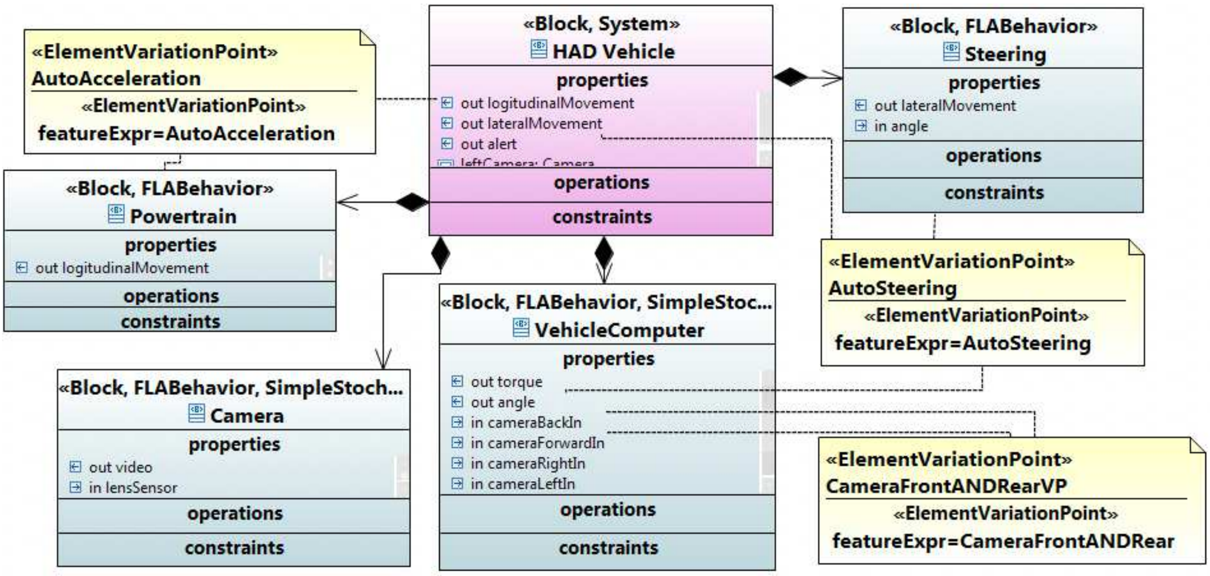


Figure 31 - The HAD Vehicle Block Definition Diagram with CRITVAR annotations

different manufacturers and choose between a Front and Rear, Side Cameras, or both camera settings.

The HAD Vehicle architecture was specified using SysML Block Definition (BDD) and Internal Block (IBD) Diagrams. Figure 31 shows an excerpt of the base HAD Vehicle BDD containing its variation points and model elements. The dotted lines denote the *annotatedElements* of the AutoAcceleration *ElementVariationPoint* is activated whenever the *AutoAcceleration* feature is selected. When activated, the *AutoAcceleration* VariationPoint keeps the *Powertrain* Block, the *longitudinalMovement* FlowPort and any instances of the *Powertrain* in an *HAD Vehicle* configuration. Vehicle components safety information were specified using CHESS *SimpleStochasticBehavior* and *FLABehavior* element stereotypes (MAZZINI et al., 2016). The *SimpleStochasticBehavior* stereotype was used to describe random faults and their consequences, i.e., how random failures propagate throughout components. The *FLABehavior* stereotype is applied to describe how components react to incoming failures and failure propagation.

Figure 32 shows the base IBD of the HAD Vehicle. The components *cameraFront* and *cameraRear* are kept in the transformed model only if the feature CameraFrontANDRear is selected. Similarly, *cameraLeft* and *cameraRight* are kept if the feature CameraSides is selected.

Figure 33 shows the Vehicle Computer *SimpleStochasticBehavior* failure annotations with CRITVAR mappings. Figure 34 shows an excerpt of the pvSCL variable conditional, used to specify variability in failure annotations within the Vehicle Computer using pure::variants. The *failureOccurrence* property takes a probability distribution (e.g. linear, deterministic, weibull) and describes the probability of occurrence of a random hardware failure. Whenever the CompManufacturer1 feature is selected, the probability

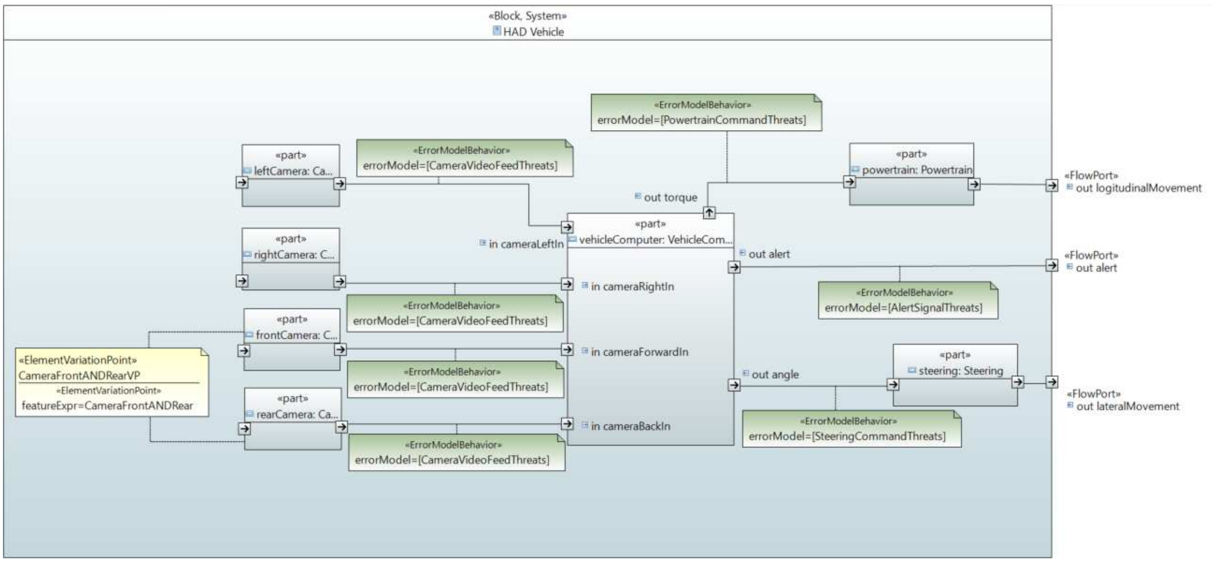


Figure 32 - The HAD Vehicle Internal Block Diagram with CRITVAR annotations

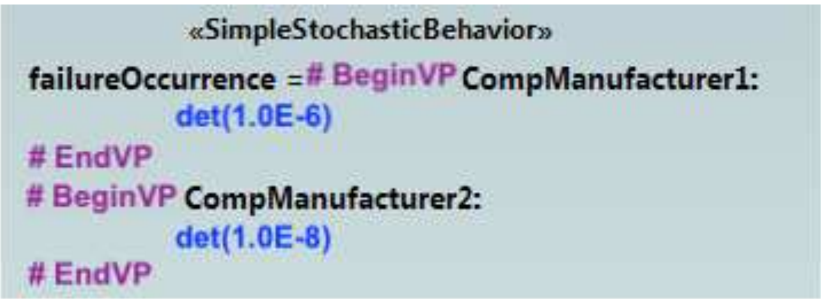
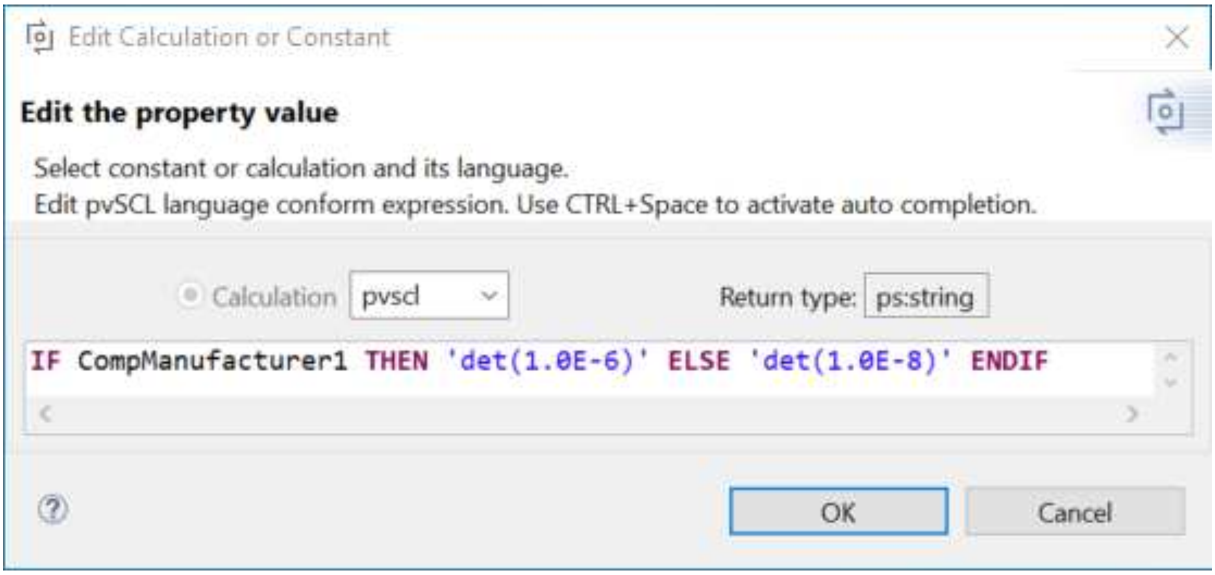

«SimpleStochasticBehavior»
failureOccurrence =# BeginVP CompManufacturer1:
det(1.0E-6)
# EndVP
# BeginVP CompManufacturer2:
det(1.0E-8)
# EndVP

Figure 33 - Vehicle Computer stochastic annotations using CRITVAR

distribution describing the Vehicle Computer *failureOcurrence* will be **det(1.0e-6)**. Otherwise, if CompManufacturer2 is selected, then the computer *failureOcurrence* is reduced to **det(1.0e-8)**.

As for deterministic failures, all components in the architecture propagate failure modes forward through their output ports if all their input ports fail. Deterministic failures and their propagation logic were defined using fptc rules via the *FLABehavior* stereotype. Since the input and output ports of the *VehicleComputer* vary depending on the variant, its fptc rules also contain variability. For example, ports *torque* and *angle* are removed from the *VehicleComputer* fptc rules if the feature Manual is selected. Figure 35 shows the pure::variants variability annotations in the *VehicleComputer* fptc annotations. Since CRITVAR automatically handles variability in fptc rules, annotating the rules with variability was not necessary as in pure::variants.

With respect to security, in the HAD Vehicle, each component poses a potential attack surface. For the threat analysis, we have defined our trust boundaries around the *VehicleComputer*, *Steering* and *Powertrain* components. Table 2 displays the model assets, their security properties and stakeholders. The assets are also identified in Figure 32 and tagged with the «ErrorModelBehavior» CHESS stereotype:



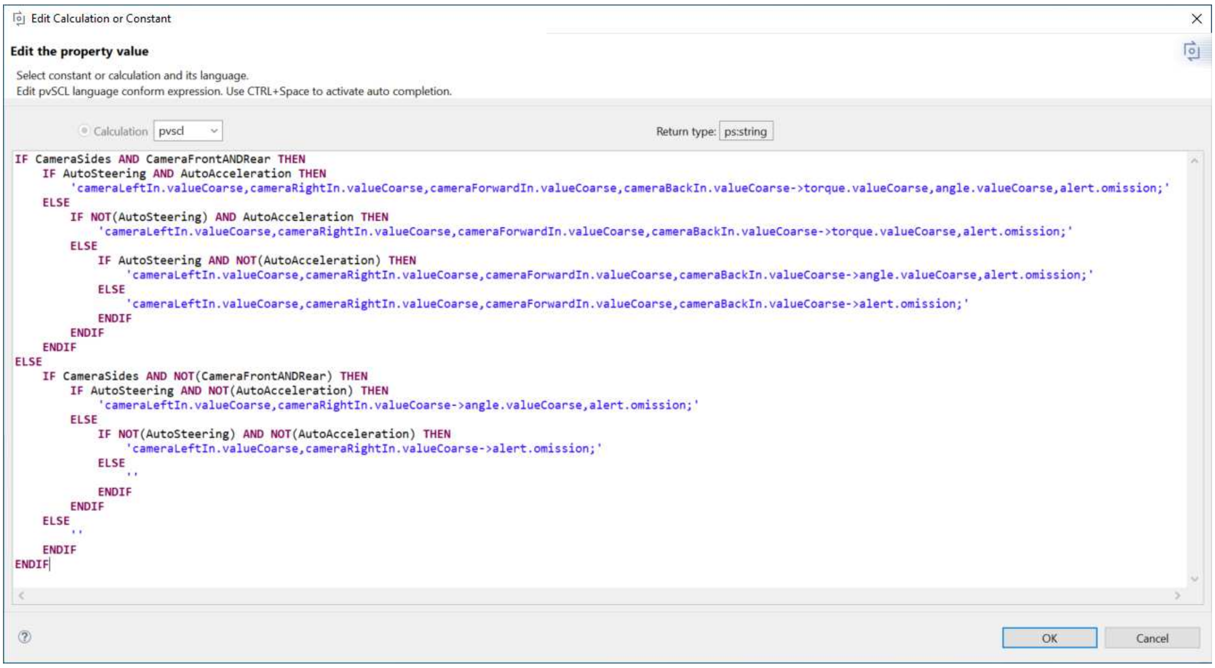
Edit the property value

Select constant or calculation and its language.
Edit pvSCL language conform expression. Use CTRL+Space to activate auto completion.

● Calculation pvscl    Return type: ps:string

```
IF CompManufacturer1 THEN 'det(1.0E-6)' ELSE 'det(1.0E-8)' ENDIF
```

Figure 34 - Vehicle Computer stochastic annotations using pure::variants



Edit the property value

Select constant or calculation and its language.
Edit pvSCL language conform expression. Use CTRL+Space to activate auto completion.

● Calculation pvscl    Return type: ps:string

```
IF CameraSides AND CameraFrontANDRear THEN
    IF AutoSteering AND AutoAcceleration THEN
        'cameraLeftIn.valueCoarse,cameraRightIn.valueCoarse,cameraForwardIn.valueCoarse,cameraBackIn.valueCoarse->torque.valueCoarse,angle.valueCoarse,alert.omission;'
    ELSE
        IF NOT(AutoSteering) AND AutoAcceleration THEN
            'cameraLeftIn.valueCoarse,cameraRightIn.valueCoarse,cameraForwardIn.valueCoarse,cameraBackIn.valueCoarse->torque.valueCoarse,alert.omission;'
        ELSE
            IF AutoSteering AND NOT(AutoAcceleration) THEN
                'cameraLeftIn.valueCoarse,cameraRightIn.valueCoarse,cameraForwardIn.valueCoarse,cameraBackIn.valueCoarse->angle.valueCoarse,alert.omission;'
            ELSE
                'cameraLeftIn.valueCoarse,cameraRightIn.valueCoarse,cameraForwardIn.valueCoarse,cameraBackIn.valueCoarse->alert.omission;'
            ENDIF
        ENDIF
    ENDIF
ELSE
    IF CameraSides AND NOT(CameraFrontANDRear) THEN
        IF AutoSteering AND NOT(AutoAcceleration) THEN
            'cameraLeftIn.valueCoarse,cameraRightIn.valueCoarse->angle.valueCoarse,alert.omission;'
        ELSE
            IF NOT(AutoSteering) AND NOT(AutoAcceleration) THEN
                'cameraLeftIn.valueCoarse,cameraRightIn.valueCoarse->alert.omission;'
            ELSE
                ''
            ENDIF
        ENDIF
    ELSE
        ''
    ENDIF
ENDIF
```

Figure 35 - Vehicle Computer fptc annotations using pure::variants

Table 2 – HAD Assets and Security Properties

| Asset | Property | Stakeholder |
|---|---|---|
| Video Feed | Confidentiality | Pedestrians, vehicle occupants |
| Video Feed | Integrity | Vehicle occupants and pedestrians |
| Video Feed | Availability | Vehicle occupants and pedestrians |
| Torque | Integrity | Vehicle occupants and pedestrians |
| Torque | Availability | Vehicle occupants and pedestrians |
| Angle | Integrity | Vehicle occupants and pedestrians |
| Angle | Availability | Vehicle occupants and pedestrians |

Table 3 – HAD Threats and Damage Scenarios

| **Threat** | **Damage Scenario** |
|---|---|
| Disclosure of Video Feed | Invasion of privacy of pedestrians and vehicle owners |
| Tampering of Video Feed | Collision with other vehicle, pedestrian;<br>Injury of vehicle occupant |
| Denial of Service of Video Feed | Collision with other vehicle, pedestrian;<br>Injury of vehicle occupant |
| Tampering of Torque | Collision with other vehicle, pedestrian;<br>Injury of vehicle occupant |
| Denial of Service of Torque | Collision with other vehicle, pedestrian;<br>Injury of vehicle occupant |
| Tampering of Angle | Collision with other vehicle, pedestrian;<br>Injury of vehicle occupant |
| Denial of Service of Angle | Collision with other vehicle, pedestrian;<br>Injury of vehicle occupant |

The video feed is transmitted from the Cameras to the *VehicleComputer* and must address the Confidentiality, Integrity and Availability security properties. The video feed may contain pedestrians' faces and vehicle plates. Thus, disclosure of the video feed to an unauthorized party, can affect the privacy of pedestrians and other vehicle owners. Delivery of corrupted or failure to deliver the video feed to the *VehicleComputer* impact the vehicle occupants, pedestrians and other vehicle owners. For example, a corrupted video feed can trigger the *VehicleComputer* to calculate the incorrect torque to the *Powertrain* or angle to the *Steering* system and therefore, cause an accident involving the vehicle occupants, other vehicles and pedestrians.

The other two assets are the Torque and Angle signals produced by the VehicleComputer. Since both assets do not contain any sensitive or personal information as the Video Feed, they should only address integrity and availability. Failure to generate a precise torque or angle signals due to a deliberate attack, e.g., caused by malware software running in the *VehicleComputer*, can lead to an accident impacting the vehicle and other vehicle occupants and pedestrians. Table 3 lists the threats and potential damage scenarios such threats can unfold. The threats were derived based on the STRIDE model (MICROSOFT, 2009).

Each asset identified in the model was mapped to an ErrorModel containing threats and their effects on safety as shown in Figure 36. The Video Feed asset starts off in an uncompromised state. A Denial of Service attack to the asset leads to the Denial of Service of the Video feed threat. Such threat produces an omission failure on the Video Feed data being transferred between the Camera and Vehicle Computer. A masquerade attack, e.g., through a process or user using a fake identity, can be used to disclose the video stream. Moreover, attackers may also explore vulnerabilities in the implementation and tamper with the video feed. Tampering of the video feed may lead to a value failure, e.g.,

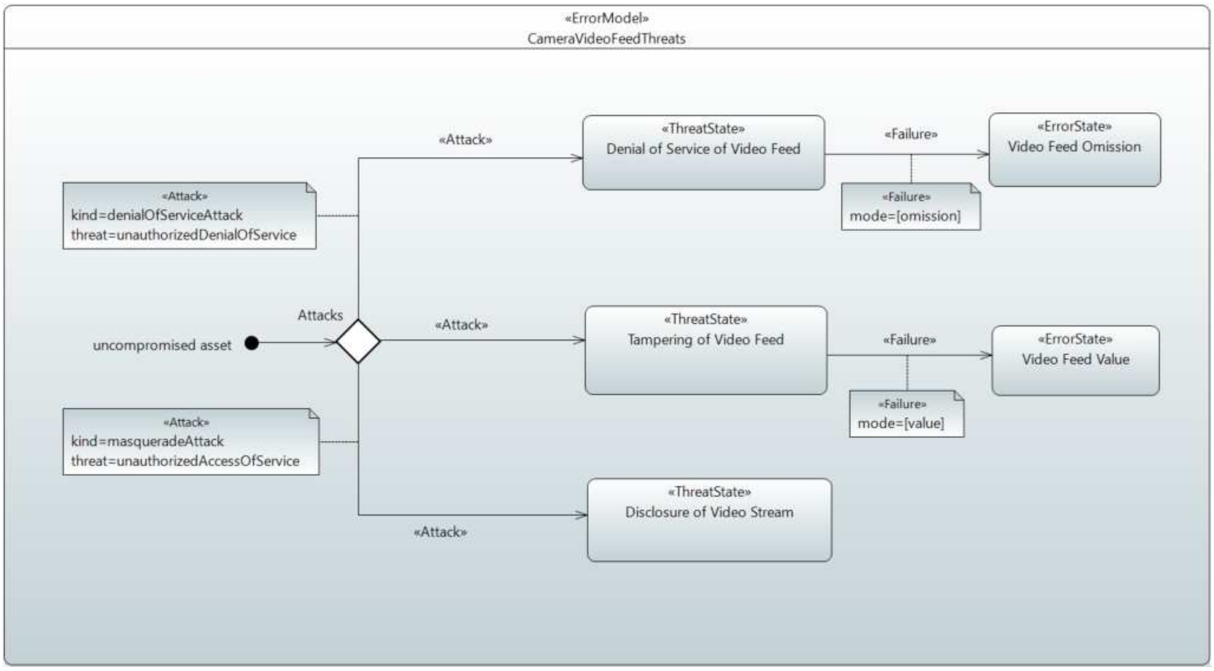


Figure 36 - Video Feed threat model

corruption or of the video feed to the Vehicle Computer.

### 7.4.2 Application Engineering Phase

The models generated in each strategy are publicly available on [4]. Table 4 displays the gathered metrics for each treatment.

Table 4 – Metric results for each considered treatment. Legend: M1 - Number of variability constructs; M2 - Number of element mappings; M3 - Number of dependability annotation mappings; M4 - Variant models are complete

| Treatment | **M1** | **M2** | **M3** | **M4** |
|---|---|---|---|---|
| CRITVAR | 28 | 4 | 2 | True |
| pure::variants | 28 | 4 | 3 | False |
| BVR | 28 | 10 | - | False |

## 7.5 DATA ANALYSIS AND RESULTS

Here, we answer the questions related to the evaluation goals, which are based on the data analysis from Table 4. We will answer to the research questions using our results from the GQM approach.

[4] https://drive.google.com/drive/u/2/folders/1IXm3YEvFm2hcnKUJVIBtJNTyODtDtAW9

### 7.5.1 RQ1 - How to represent the variability of functional safety and cybersecurity models?

In our approach, we represented variability in safety information via annotations directly into the model instead of mapping features to UML fragments in reference models. Some additional advantages from our approach regarding RQ1 are discussed below.

**Answer to Q1 - How much effort is needed for expressing variability in dependability in the domain variability model?**: The number of feature model elements (**M1:**) was the same on CRITVAR, pure::variants and BVR. Thus, all tools required the same effort in terms of the specification of the domain feature model. **Answer to Q2 - How much effort is needed to map variability constructs to dependability artifacts?**: Compared to BVR, CRITVAR and pure::variants maintain a lower number of mappings between features and model elements, i.e., *M3*. Regarding dependability annotations, CRITVAR requires a lower number of mappings when compared to pure::variants thanks to the CRITVAR MTE FLA extension. BVR does not support the specification of mappings between domain features and fine grained dependability information.

We conclude that CRITVAR combines the positive aspects from BVR and pure::variants with respect to mapping domain features to model elements. At the same time, CRITVAR also enables variability specification at a more fine-grained level compared to pure::variants. CRITVAR reduces the number of mappings between variability constructs and model elements when compared to BVR and the number of mappings between features and annotations when compared to pure::variants.

### 7.5.2 RQ2 - How derive product safety information from a variability model?

Our novel transformation approach produces a correct and complete system model containing all the necessary dependability information necessary to perform higher-level dependability analysis and according to the product requirements. We do this using the system model with variability annotations as outcome of RQ1 which is given as input for the transformation engine. An additional outcome of this RQ is discussed below.

**Answer to Q3 - How correct is the derived product configuration model with respect to feature selection?**: CRITVAR ensures the generation of complete SysML models with dependability annotations. Therefore, the HAD Vehicle SysML models derived by CRITVAR is ready for product-specific dependability analysis. Since BVR does not support variability specification in dependability annotations, BVR product configuration models should be annotated after the execution of BVR engine prior to dependability analysis.

The pure::variants transformation engine does not automatically handle element dependencies when transforming the model like the CRITVAR MTE does. This requires the product line engineer to manually tag all element dependencies prior to transformation, e.g. connector linking two component ports that vary, in order to generate a complete model. However, for certain SysML constructs, e.g., Parts of a Block being trimmed out of the product model, the pure::variants transformation engine fails when removing a dependency of a block that has been previously removed by the engine itself. Therefore, certain model elements, shall be removed manually after model transformation.

## 7.6 THREATS TO VALIDITY

The validity of the analysis is subject to construct, internal and external threats. We identified the following threats to the construct validity: **(i)** the possibility the selected metrics are not complete to generalize the results. We mitigated this threat by providing metrics *M2 - Number of element mappings* and M3 - Number of dependability annotation mappings to quantify the effort to map variability constructs to both model elements and dependability properties attached to model elements. We have also considered the completeness of the product models generated by each tool through *M4 - Variant models are complete*; **ii)** the evaluation is limited to a single object, the HAD Vehicle model. We mitigate this threat by choosing a case study that combines both system design with dependability analysis using modern model-based techniques. We recognize that this threat could affect the generalization of our results.

Regarding the internal validity, since we did not consider the time users took to map variability constructs to functional safety information, the metric set may not be complete enough to fully demonstrate the reduction in the gap between variability constructs and dependability information. In order to mitigate this threat, we used a metric to quantify the user effort in terms of the number of required mappings to model elements and dependability annotations. We also intend to further conduct an experimental study with users to assess efficiency of CRITVAR-ML in supporting the specification of mappings between features and safety artifacts.

We identified the following external validity threats: *interaction of selection and treatment*: **i)** the subject considered in the evaluation could not be a fully representative of the population we aimed to generalize the results. We tried to mitigate this threat by choosing a subject who had previous experience with the application of product line engineering activities and all the considered treatments. When it comes to the *interaction of setting and treatment ii)* one of the treatments considered in the evaluation, i.e., BVR, may not fully represent solutions that are being currently adopted in the industry. Therefore, we have also considered pure::variants, a commercial variability management tool widely used in the industry.

# 8 CONCLUSIONS AND FUTURE WORK

In this work, we presented CRITVAR: a modeling language, model transformation engine and process to address variability in safety and cybersecurity model artifacts. CRITVAR has been driven by the challenges and gaps identified in existing approaches: **CH1)** Integrating model-based safety and cybersecurity co-analyses within product line processes; **CH2)** Reducing the gap between variability constructs and finer-grained functional safety and cybersecurity analysis information expressed as annotations attached to MOF-compliant system models; **CH3)** Enabling the systematic reuse of system models enriched with functional safety and cybersecurity analysis information; and **CH4)** Ensuring the derivation of correct and complete system models enriched with functional safety and cybersecurity information from a variability (feature) model.

As contributions, this work provides the following:

- **CRITVAR-ML:** A variability realization modeling language built upon CVL to support the mapping of domain variability into dependability models at coarse and fine-grained levels;
- **CRITVAR MTE:** A model transformation tool compatible with MOF-compliant models, e.g., UML and SysML, and state-of-the-art variability management solutions, i.e., pure::variants and BVR;
- **CRITVAR Process:** An extension of functional safety, cybersecurity and product line processes to support the management of variability in dependability models.

As benefits, CRITVAR successfully addresses challenges CH1 through CH4 revisited earlier in this chapter. **CH1)** Integrating model-based safety and cybersecurity co-analyses within product line processes, is addressed by the CRITVAR process which provides a systematic way to address safety, security and variability concerns in alignment with system-level threat and safety analysis activities from the ISO 26262 and SAE/ISO 21434.

As for **CH2)** Reducing the gap between variability constructs and finer-grained functional safety and cybersecurity analysis information expressed as annotations attached to MOF-compliant system models, CRITVAR-ML reduces the gap between variability constructs and finer-grained dependability information. Such reduction can be observed in two fronts: i) ability to specify variability within finer-grained model dependability information, e.g., FPTC annotations; and ii) reduction in the number of mappings between features and model artifacts when compared to existing variability management solutions, i.e., pure::variants and BVR.

**CH3)** Enabling the systematic reuse of system models enriched with functional safety and cybersecurity analysis information is addressed again by the CRITVAR process.

CRITVAR-ML and CRITVAR-MTE are both enablers to the systematic reuse of dependability models. CRITVAR-ML supports the CRITVAR process by providing a notation for variability resolution. CRITVAR-MTE provides a tool for performing the realization of variability and product-specific dependability analysis. Moreover, CRITVAR-MTE also ensures the derivation of complete and correct dependability models thus, addressing *CH4)* Ensuring the derivation of correct and complete system models enriched with functional safety and cybersecurity information from a variability (feature) model.

However, as in other approaches evaluated in this work, CRITVAR also suffers from a limitations. Current limitations of CRITVAR can be classified into two groups: limitations with respect to compatibility and; limitations with respect to evaluation.

Limitations with respect to compatibility in CRITVAR include the: lack of support for additional model-based solutions, e.g., Simulink, SCADE Architect, Medini Analyze, AADL, and; additional variability modeling solutions, e.g., FeatureIDE. However, these limitations can be easily addressed due the extensible nature of CRITVAR-MTE. New adaptors can be implemented and integrated into CRITVAR-MTE to add support for new model-based and variability management solutions.

Limitations with respect to evaluation include: lack of evaluation to understand the benefits of the CRITVAR-ML notation over similar variability realization languages and solutions, e.g., BVR, pure::variants and; lack of evaluation of the CRITVAR process in an industrial context.

Given the aforementioned limitations on the compatibility of CRITVAR, we intend to extend the compatibility of CRITVAR by: **i)** Creating adaptors to address Matlab Simulink models enriched with dependability annotations from HiP-HOPS; **ii)** Creating adaptors to address AADL models enriched with Error Annex dependability information; **iii)** Creating adaptors to support additional variability management tools such as FeatureIDE. Moreover, based on the limitations with respect to the evaluation of CRITVAR, we intend to: **iv)** Perform a comparative evaluation of the CRITVAR-ML notation to measure its efficiency over state-of-the-art realization modeling notations such as the one implemented in pure::variants and; **v)** Perform an evaluation of the CRITVAR process in an industrial context.

# References

ABAL, Iago et al. Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. **ACM Trans. Softw. Eng. Methodol.**, Association for Computing Machinery, New York, NY, USA, v. 26, n. 3, Jan. 2018. ISSN 1049-331X. DOI: 10.1145/3149119. Available from: <https://doi.org/10.1145/3149119>.

ABELE, Andreas et al. EPM - A prototype tool for variability management in component hierarchies. **ACM International Conference Proceeding Series**, v. 2, p. 246–249, 2012.

ACHER, Mathieu et al. Product Lines Can Jeopardize Their Trade Secrets. In: PROCEEDINGS of the 2015 10th Joint Meeting on Foundations of Software Engineering. Bergamo, Italy: Association for Computing Machinery, 2015. (ESEC/FSE 2015), p. 930–933. ISBN 9781450336758. DOI: 10.1145/2786805.2803210. Available from: <https://doi.org/10.1145/2786805.2803210>.

ADLER, Rasmus et al. Integration of Component Fault Trees into the UML. In: DINGEL, Juergen; SOLBERG, Arnor (Eds.). **Models in Software Engineering**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. P. 312–327. ISBN 978-3-642-21210-9.

ALBERTS, C. J. et al. **Operationally critical threat, asset, and vulnerability evaluation (OCTAVE) framework, Version 1.0**. 1999.

AMASS. **Baseline and requirements for multi-concern assurance D4.1**. 2018.

ANDERSON, J. **Computer Security Planning Study. Technical Report ESD-TR-73-51. Air Force Electronic System Division.** 1972.

ANDREWS, J. D.; DUNNETT, S. J. Event-tree analysis using binary decision diagrams. **IEEE Transactions on Reliability**, v. 49, n. 2, p. 230–238, 2000. DOI: 10.1109/24.877343.

ANSYS. **Ansys Medini Analyze: Safety-Critical Electronic Software Analysis**. 2020.

ANSYS. **Medini Analyze Tool**. 2020. https://www.ansys.com/products/systems/ansys-medini-analyze.

AUSTON. **Bombardier Railway Solutions**. 2021. Available from: <https://rail.bombardier.com/en.html>.

AUTOSAR. **UML Profile for AUTOSAR**. 2006. Available from: <https://www.autosar.org/fileadmin/user_upload/standards/classic/2-0/AUTOSAR_UML_Profile.pdf>.

AVIZIENIS, A.; LAPRIE, J.-C.; RANDELL, B., et al. Basic concepts and taxonomy of dependable and secure computing. **IEEE Transactions on Dependable and Secure Computing**, v. 1, n. 1, p. 11–33, 2004. DOI: 10.1109/TDSC.2004.2.

AVIZIENIS, Algirdas; LAPRIE, Jean-Claude; BRIAN, Randell. Fundamental Concepts of Dependability, 2000. Available from: <https://www.cs.rutgers.edu/~rmartin/teaching/spring03/cs553/readings/avizienis00.pdf>.

AZEVEDO, L. P. S. **Scalable Allocation of Safety Integrity Levels in Automotive Systems**. 2015.

BASILI, Victor R; CALDIERA, Gianluigi; ROMBACH, H Dieter. The goal question metric approach. **Encyclopedia of Software Engineering**, v. 2, p. 528–532, 1994. ISSN <null>.

BEHRINGER, Benjamin; PALZ, Jochen; BERGER, Thorsten. PEoPL: Projectional Editing of Product Lines. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). IEEE, May 2017. P. 563–574. ISBN 978-1-5386-3868-2. DOI: 10.1109/ICSE.2017.58. Available from: <http://ieeexplore.ieee.org/document/7985694/>.

BEUCHE, Danilo; SCHULZE, Michael; DUVIGNEAU, Maurice. When 150% is too much: Supporting product centric viewpoints in an industrial product line. **ACM International Conference Proceeding Series**, 16-23-Sept, p. 262–269, 2016.

BLOM, Hans et al. East-Adl. **International Journal of System Dynamics Applications**, v. 5, n. 3, p. 1–20, 2016. ISSN 2160-9772.

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **The Unified Modeling Language User Guide (2nd Edition)**. Addison-Wesley Professional, 2005. ISBN 0321267974.

BOSCO, João et al. Customizing the common variability language semantics for your domain models. In: September. PROCEEDINGS of the VARiability for You Workshop: Variability Modeling Made Useful for Everyone, VARY 2012. 2012. P. 3–8. ISBN 9781450318099. DOI: 10.1145/2425415.2425417.

BOTTERWECK, Goetz; POLZER, Andreas; KOWALEWSKI, Stefan. Using higher-order transformations to derive variability mechanism for embedded systems. In: CEUR Workshop Proceedings. 2009. v. 507, p. 107–121.

BOZZANO, M.; VILLAFIORITA, Adolfo. **Design and Safety Assessment of Critical Systems**. 1st. USA: Auerbach Publications, 2010. ISBN 1439803315.

BRAGA, R.; BRANCO, K., et al. The ProLiCES Approach to Develop Product Lines for Safety-Critical Embedded Systems and its Application to the Unmanned Aerial Vehicles Domain. **CLEI Electron. J.**, v. 15, 2012.

BRAGA, Rosana T. V.; TRINDADE, Onofre, et al. Incorporating certification in feature modelling of an unmanned aerial vehicle product line. In: PROCEEDINGS of the 16th International Software Product Line Conference - SPLC '12. Association for Computing Machinery (ACM), Sept. 2012. P. 1–10. DOI: 10.1145/2362536.2362570.

BRAGA, Rosana T. V.; TRINDADE JUNIOR, Onofre, et al. Adapting a Software Product Line Engineering Process for Certifying Safety Critical Embedded Systems. In: ORTMEIER, Frank; DANIEL, Peter (Eds.). **Computer Safety, Reliability, and Security**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. P. 352–363. ISBN 978-3-642-33678-2.

BRESSAN, L.; OLIVEIRA, A. L.; CAMPOS, F.; CAPILLA, R. A variability modeling and transformation approach for safety-critical systems. In: VAMOS '21: 15th International Working Conference on Variability Modelling of Software-Intensive Systems. 2021. DOI: `10.1145/3442391.3442398`.

BRESSAN, L.; OLIVEIRA, A. L. de; CAMPOS, F.; PAPADOPOULOS, Y., et al. An Integrated Approach to Support the Process-Based Certification of Variant-Intensive Systems. In: ZELLER, Marc; HÖFIG, Kai (Eds.). **Model-Based Safety and Assessment**. Cham: Springer International Publishing, 2020. P. 179–193. ISBN 978-3-030-58920-2.

BRESSAN, Lucas; BRAGA, Regina, et al. An ontology-based approach to support the certification of Safety-Critical Software Product Lines. In: ANAIS do XLVII Seminário Integrado de Software e Hardware. Cuiabá: SBC, 2020. P. 81–92. DOI: `10.5753/semish.2020.11319`. Available from: <`https://sol.sbc.org.br/index.php/semish/article/view/11319`>.

BRESSAN, Lucas; OLIVEIRA, Andre L. de; CAMPOS, Fernanda. An Approach to Support Variant Management on Safety Analysis using CHESS Error Models. In: 2020 16th European Dependable Computing Conference (EDCC). IEEE, Sept. 2020. P. 135–142. ISBN 978-1-7281-8936-9. DOI: `10.1109/EDCC51268.2020.00030`.

BRESSAN, Lucas; PIOLI, Laércio, et al. An Approach to Support the Design and the Dependability Analysis of High Performance I/O Intensive Distributed Systems. In: BAROLLI, Leonard et al. (Eds.). **Advances on P2P, Parallel, Grid, Cloud and Internet Computing**. Cham: Springer International Publishing, 2021. P. 29–40. ISBN 978-3-030-61105-7.

BYVAIKOV, ME et al. Experience from design and application of the top-level system of the process control system of nuclear power-plant. **Automation and Remote Control**, Springer, v. 67, n. 5, p. 735–747, 2006.

CAPELLA. **Capella Model-Based Systems Engineering Tool**. 2020. Available from: <`https://www.eclipse.org/capella/`>.

CHEMICAL INDUSTRIES ASSOCIATION. **A Guide to Hazard and Operability Studies, Chemical Industries Association**. 1977.

CLAUSS, M. **Modeling variability with UML**. 2001.

CLEMENTS, Paul; NORTHROP, Linda. **Software Product Lines: Practices and Patterns**. Addison-Wesley Professional, 2001.

CMU. **Open Source AADL Tool Environment (OSATE)**. 2020. Accessed on 19.01.2021. Available from: <`osate.org`>.

COLEMAN, Joey W. et al. COMPASS tool vision for a system of systems collaborative development environment. In: PROCEEDINGS - 2012 7th International Conference on System of Systems Engineering, SoSE 2012. 2012. P. 451–456. ISBN 9781467329750. DOI: `10.1109/SYSoSE.2012.6384150`.

COMMISSION OF THE EUROPEAN COMMUNITIES. **Information Technology Security Evaluation Criteria (ITSEC)**. Office for official publications of the european communites, 1991.

CZARNECKI, Krzysztof; ANTKIEWICZ, Michał. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: 2005. P. 422–437. DOI: `10.1007/11561347{\_}28`. Available from: <`http://link.springer.com/10.1007/11561347_28`>.

DEGUEULE, Thomas et al. Tooling support for variability and architectural patterns in systems engineering. **ACM International Conference Proceeding Series**, 20-24-July, p. 361–364, 2015.

DEHLINGER, Josh; LUTZ, Robyn R. Software fault tree analysis for product lines. In: PROCEEDINGS of IEEE International Symposium on High Assurance Systems Engineering. 2004. v. 8, p. 12–21.

DEIS. **Open Dependability Exchange (ODE) Profile V2**. 2020. Available from: <`https://www.deis-project.eu`>.

DELANGE, J. **Architecture Analysis  Design Language. Number SAE AS5506C. SAE International**. 2016. Accessed on 12.09.2021. Available from: <`https://www.sae.org/standards/content/as5506/`>.

DOBAJ, Jürgen et al. Towards Integrated Quantitative Security and Safety Risk Assessment. In: ROMANOVSKY, Alexander et al. (Eds.). **Computer Safety, Reliability, and Security**. Cham: Springer International Publishing, 2019. P. 102–116. ISBN 978-3-030-26250-1.

DOMIS, Dominik; ADLER, Rasmus; BECKER, Martin. Integrating Variability and Safety Analysis Models Using Commercial UML-Based Tools. In: (SPLC ’15), p. 225–234. ISBN 9781450336130. DOI: `10.1145/2791060.2791088`. Available from: <`https://doi.org/10.1145/2791060.2791088`>.

DOMIS, Dominik; ADLER, Rasmus; BECKER, Martin. Integrating variability and safety analysis models using commercial UML-based tools. **ACM International Conference Proceeding Series**, 20-24-July, p. 225–234, 2015.

DORDOWSKY, Frank; BRIDGES, Richard; TSCHOPE, Holger. Implementing a software product line for a complex avionics system. **Proceedings - 15th International Software Product Line Conference, SPLC 2011**, p. 241–250, 2011.

EAST-ADL ASSOCIATION. **EAST-ADL Domain Model Specification version V2.1.12**. 2013. Available from: <https://www.east-adl.info/Specification/V2.1.12/EAST-ADL-Specification_V2.1.12.pdf>.

ECLIPSE FOUNDATION. **Eclipse Papyrus Modeling environment**. 2017. Available from: <https://www.eclipse.org/papyrus/>.

EMBRAER. **Embraer Defense Systems**. 2021. Available from: <https://defense.embraer.com/global/en/defense-systems>.

ETSI. **TS 102 165-1: Telecommunications and internet converged services and protocols for advanced networking (tispan); methods and protocols; part 1: Method and proforma for threat, risk, vulnerability analysis**. 2011. Available from: <https://www.etsi.org/deliver/etsi_ts/102100_102199/10216501/04.02.03_60/ts_10216501v040203p.pdf>.

FEILER, P. Architecture analysis and design language (aadl) annex volume 3: Annex e: Error model v2 annex. **Number SAE AS5506/3 (Draft) in SAE Aerospace Standard. SAE International**, 2013.

FENELON, Peter; MCDERMID, John A. An integrated tool set for software safety analysis. **The Journal of Systems and Software**, v. 21, n. 3, p. 279–290, 1993. ISSN 01641212. DOI: 10.1016/0164-1212(93)90029-W.

FENG, Qian; LUTZ, Robyn R. Bi-directional safety analysis of product lines. **Journal of Systems and Software**, v. 78, n. 2, p. 111–127, 2005. ISSN 01641212.

FLORES, Leandro; OLIVEIRA, Edson. SmartyModeling: an Environment for Engineering UML-based Software Product Lines. In: VAMOS '21: 15th International Working Conference on Variability Modelling of Software-Intensive Systems. 2021. ISBN 9781450388245. DOI: https://doi.org/10.1145/3442391.3442397.

FOSTER, I. et al. Fast and Vulnerable: A Story of Telematic Failures. In: (WOOT'15), p. 15.

GALLINA, Barbara; JAVED, Muhammad Atif, et al. A model-driven dependability analysis method for component-based architectures. In: PROCEEDINGS - 38th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2012. 2012. P. 233–240. ISBN 9780769547909. DOI: 10.1109/SEAA.2012.35.

GALLINA, Barbara; MONTECCHI, Leonardo, et al. Multiconcern, Dependability-Centered Assurance Via a Qualitative and Quantitative Coanalysis. **IEEE Software**, v. 39, n. 4, p. 39–47, 2022. DOI: 10.1109/MS.2022.3167370.

GALLINA, Barbara; PUNNEKKAT, Sasikumar. FI4FA: A Formalism for Incompletion, Inconsistency, Interference and Impermanence Failures' Analysis. In: 2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications. IEEE, Aug. 2011. P. 493–500. ISBN 978-1-4577-1027-8. DOI: `10.1109/SEAA.2011.80`. Available from: <`http://ieeexplore.ieee.org/document/6068389/`>.

GOMAA, Hassan. **Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures**. USA: Addison Wesley Longman Publishing Co., Inc., 2004. ISBN 0201775956.

GÓMEZ, Carolina; LIGGESMEYER, Peter; SUTOR, Ariane. Variability Management of Safety and Reliability Models: An Intermediate Model towards Systematic Reuse of Component Fault Trees. In: SAFECOMP 2010: Computer Safety, Reliability, and Security. 2010. P. 28–40. DOI: `10.1007/978-3-642-15651-9{\_}3`.

GREENGARD, Samuel. Automotive Systems Get Smarter. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 58, n. 10, p. 18–20, Sept. 2015. ISSN 0001-0782. DOI: `10.1145/2811286`. Available from: <`https://doi.org/10.1145/2811286`>.

GROHER, Iris; VÖLTER, Markus. Expressing Feature-Based Variability in Structural Models. **Workshop on Managing Variability for Software Product Lines**, 2007.

GRÖNNINGER, Hans et al. Modeling Variants of Automotive Systems using Views, 2014. Available from: <`http://arxiv.org/abs/1409.6629`>.

GURP, J. van; BOSCH, J.; SVAHNBERG, M. On the notion of variability in software product lines. In: PROCEEDINGS Working IEEE/IFIP Conference on Software Architecture. 2001. P. 45–54. DOI: `10.1109/WICSA.2001.948406`.

GURP, J. van; BOSCH, J.; SVAHNBERG, M. On the notion of variability in software product lines. In: PROCEEDINGS Working IEEE/IFIP Conference on Software Architecture. IEEE Comput. Soc. P. 45–54. ISBN 0-7695-1360-3. DOI: `10.1109/WICSA.2001.948406`. Available from: <`http://ieeexplore.ieee.org/document/948406/`>.

HAAS, P.J.; SHEDLER, G.S. Stochastic Petri net representation of discrete event simulations. **IEEE Transactions on Software Engineering**, v. 15, n. 4, p. 381–393, Apr. 1989. ISSN 0098-5589. DOI: `10.1109/32.16599`. Available from: <`http://ieeexplore.ieee.org/document/16599/`>.

HABER, Arne et al. First-class variability modeling in Matlab/Simulink. **ACM International Conference Proceeding Series**, p. 11–18, 2013.

HABLI, I.; KELLY, T.; HOPKINS, I. Challenges of Establishing a Software Product Line for an Aerospace Engine Monitoring System. In: 11TH International Software Product Line Conference (SPLC 2007). 2007. P. 193–202. DOI: `10.1109/SPLINE.2007.37`.

HAIDER, Zulqarnain; GALLINA, Barbara; MORENO, Enrique Zornoza. FLA2FT: Automatic Generation of Fault Tree from ConcertoFLA Results. In: PROCEEDINGS - 2018 3rd International Conference on System Reliability and Safety, ICSRS 2018. Institute of Electrical and Electronics Engineers Inc., Apr. 2019. P. 176–181. ISBN 9781728102382.

HAUGEN, Ø; MØLLER-PEDERSEN, B, et al. Adding Standardized Variability to Domain Specific Languages. In: 2008 12th International Software Product Line Conference. 2008. P. 139–148.

HAUGEN, Øystein. **D4.1 BVR - The Language**. 2014.

HAUGEN, Øystein. **VARIES: VARiability In safety-critical Embedded Systems, D4.2 BVR - The Language**. 2014.

HAUGEN, Øystein; ØGÅRD, Ommund. BVR – Better Variability Results. In: AMYOT, Daniel; CASAS, Pau i; MUSSBACHER, Gunter (Eds.). **System Analysis and Modeling: Models and Reusability**. Cham: Springer International Publishing, 2014. P. 1–15. ISBN 978-3-319-11743-0. DOI: 10.1007/978-3-319-11743-0{_}1. Available from: <http://link.springer.com/10.1007/978-3-319-11743-0_1>.

HAUGEN, Øystein; WASOWSKI, Andrzej; CZARNECKI, Krzysztof. CVL .- Common variability language. **ACM International Conference Proceeding Series**, v. 2, August, p. 266–267, 2012.

HAUGEN, Øystein; WASOWSKI, Andrzej; CZARNECKI, Krzysztof. CVL: common variability language. In: PROCEEDINGS of the 16th International Software Product Line Conference on - SPLC '12 -volume 1. New York, New York, USA: ACM Press, 2012. P. 266. ISBN 9781450310956. DOI: 10.1145/2364412.2364462. Available from: <http://dl.acm.org/citation.cfm?doid=2364412.2364462>.

HORCAS, Jose-Miguel; CORTIÑAS, Alejandro, et al. Integrating the common variability language with multilanguage annotations for web engineering. In: PROCEEDINGS of the 22nd International Systems and Software Product Line Conference - Volume 1. New York, NY, USA: ACM, Sept. 2018. P. 196–207. ISBN 9781450364645. DOI: 10.1145/3233027.3233049. Available from: <https://dl.acm.org/doi/10.1145/3233027.3233049>.

HORCAS, Jose-Miguel; PINTO, Mónica; FUENTES, Lidia. An Aspect-Oriented Model Transformation to Weave Security using CVL. In: 2014 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD). 2014. P. 138–15.

HORCAS, Jose-Miguel; PINTO, Mónica; FUENTES, Lidia. Proceedings of the 21st International Systems and Software Product Line Conference - Volume B. In: Cvl. v. 2, p. 32–37. ISBN 9781450351195. DOI: 10.1145/3109729.3109749. Available from: <https://dl.acm.org/doi/10.1145/3109729.3109749>.

IEC. IEC 61508 - Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related System. In.

IEC. **Security for industrial automation and control systems - Part 4-1: Secure product development lifecycle requirements**. 2018.

INTECS. **CHESS Modelling Language UML / MARTE / SysML profile**. 2020. Available from: <`https://www.eclipse.org/chess/publis/CHESSMLprofile.pdf`>.

ISO. **ISO/SAE 21434: Road vehicles — Cybersecurity engineering**. 2021.

ISO. **Road vehicles – Functional safety**. ISO, Geneva, Switzerland, 2011.

ISO. **Road vehicles – Functional safety**. ISO, Geneva, Switzerland, 2018.

JAVED, Muhammad Atif; GALLINA, Barbara. Safety-oriented process line engineering via seamless integration between EPF composer and BVR tool. In: p. 23–28.

JAVED, Muhammad Atif; GALLINA, Barbara; CARLSSON, Anna. Towards variant management and change impact analysis in safety-oriented process-product lines. In: p. 2372–2375.

JOSHI, A. et al. A proposal fr model-based safety analysis. In: January. 24TH Digital Avionics Systems Conference. IEEE, 2005. v. 2, p. 2–1. ISBN 0-7803-9307-4. DOI: `10.1109/DASC.2005.1563469`. Available from: <`http://ieeexplore.ieee.org/document/1563469/`>.

KAESSMEYER, Michael; MONCADA, David Santiago Velasco; SCHURIUS, Markus. Evaluation of a systematic approach in variant management for safety-critical systems development. In: PROCEEDINGS - IEEE/IFIP 13th International Conference on Embedded and Ubiquitous Computing, EUC 2015. Institute of Electrical and Electronics Engineers Inc., Dec. 2015. P. 35–43. ISBN 9781467382991.

KANG, K. C. et al. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. J. C. Baltzer AG, Science Publishers, USA, v. 5, n. 1, p. 143–168, 1998. ISSN 1022-7091.

KANG, Kyo C et al. **Feature-Oriented Domain Analysis (FODA) Feasibility Study**. 1990.

KÄSSMEYER, Michael; SCHULZE, Michael; SCHURIUS, Markus. A process to support a systematic change impact analysis of variability and safety in automotive functions. In: ACM International Conference Proceeding Series. 2015. 20-24-July, p. 235–244. ISBN 9781450336130. DOI: `10.1145/2791060.2791079`.

KENNER, Andy; DASSOW, Stephan, et al. Using Variability Modeling to Support Security Evaluations: Virtualizing the Right Attack Scenarios. In: PROCEEDINGS of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems. Magdeburg, Germany: Association for Computing Machinery, 2020. (VAMOS '20). ISBN 9781450375016. DOI: 10.1145/3377024.3377026. Available from: <https://doi.org/10.1145/3377024.3377026>.

KENNER, Andy; MAY, Richard, et al. Safety, Security, and Configurable Software Systems: A Systematic Mapping Study. In: PROCEEDINGS of the 25th ACM International Systems and Software Product Line Conference - Volume A. New York, NY, USA: Association for Computing Machinery, 2021. P. 148–159. ISBN 9781450384698. Available from: <https://doi.org/10.1145/3461001.3471147>.

KLETZ, T. A. **HAZOP and Hazan**. 4th ed.: Taylor  Francis, 1999.

KNIGHT, J.C. Safety critical systems: challenges and directions. In: PROCEEDINGS of the 24th International Conference on Software Engineering. ICSE 2002. 2002. P. 547–550.

LACKNER, Hartmut; SCHLINGLOFF, Bernd Holger. **Advances in Testing Software Product Lines**. 1. ed.: Elsevier Inc., 2017. v. 107, p. 157–217. Available from: <http://dx.doi.org/10.1016/bs.adcom.2017.07.001>.

LANCELOTI, Leandro A. et al. SMartyParser: A XMI parser for UML-based software product line variability models. In: i. VAMOS '13: Proceedings of the Seventh International Workshop on Variability Modelling of Software-Intensive Systems. 2013. P. 1–5. ISBN 9781450315418.

LAUTENBACH, Aljoscha; ISLAM, Mafijul. **Security models**. 2014.

LEE, K.; KANG, Kyo C. Usage Context as Key Driver for Feature Selection. In: BOSCH, Jan; LEE, Jaejoon (Eds.). **Software Product Lines: Going Beyond**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. P. 32–46. ISBN 978-3-642-15579-6.

LEVESON, N. **Safeware: System Safety and Computers**. Addison-Wesley, 1995.

LIPNER, S.; ANDERSON, R. **CIA history**. Personal communication, 2018.

LISAGOR, O; MCDERMID, J; PUMFREY, D. Towards a Practicable Process for Automated Safety Analysis. In: 24TH International System Safety Conference (ISSC). 2006. P. 596–607.

LISAGOR, O.; KELLY, T.; NIU, R. Model-based safety assessment: Review of the discipline and its challenges. In: THE Proceedings of 2011 9th International Conference on Reliability, Maintainability and Safety. 2011. P. 625–632. DOI: 10.1109/ICRMS.2011.5979344.

LISOVA, Elena; ŠLJIVO, Irfan; ČAUŠEVIĆ, Aida. Safety and Security Co-Analyses: A Systematic Literature Review. **IEEE Systems Journal**, IEEE, v. 13, n. 3, p. 2189–2200, 2019. ISSN 19379234.

LIU, Jing; DEHLINGER, Josh; LUTZ, Robyn. Safety analysis of software product lines using state-based modeling. **Journal of Systems and Software**, v. 80, n. 11, p. 1879–1892, Nov. 2007. ISSN 01641212.

MACGREGOR, John; BURTON, Simon. Challenges in Assuring Highly Complex, High Volume Safety-Critical Software. In: GALLINA, Barbara et al. (Eds.). **Computer Safety, Reliability, and Security**. Springer International Publishing, 2018. P. 252–264. ISBN 978-3-319-99229-7.

MACHER, G.; SCHMITTNER, C.; VELEDAR, O., et al. ISO/SAE DIS 21434 Automotive Cybersecurity Standard - In a Nutshell. In: CASIMIRO, António et al. (Eds.). **Computer Safety, Reliability, and Security. SAFECOMP 2020 Workshops**. Cham: Springer International Publishing, 2020. P. 123–135. ISBN 978-3-030-55583-2.

MACHER, G.; SPORER, H., et al. SAHARA: A security-aware hazard and risk analysis method. In: 2015 Design, Automation Test in Europe Conference Exhibition (DATE). 2015. P. 621–624. DOI: 10.7873/DATE.2015.0622.

MACHER, Georg; SCHMITTNER, Christoph; ARMENGAUD, Eric, et al. **Integration of security in the development lifecycle of dependable automotive CPS**. 2017. P. 383–423. ISBN 9781522528463. DOI: 10.4018/978-1-5225-2845-6.ch015.

MALHOTRA, M.; TRIVEDI, K.S. Dependability modeling using Petri-nets. **IEEE Transactions on Reliability**, v. 44, n. 3, p. 428–440, 1995. DOI: 10.1109/24.406578.

MATHWORKS. **Simulink - Simulation and Model-Based Design**. 2021. Accessed on 19.01.2021. Available from: <https://www.mathworks.com/products/simulink.html>.

MAZZINI, Silvia et al. CHESS: An open source methodology and toolset for the development of critical systems. **CEUR Workshop Proceedings**, v. 1835, p. 59–66, 2016. ISSN 16130073.

MCGRAW, G. **Software Security: Building Security in**. Addison-Wesley, 2006. (Addison-Wesley professional computing series). ISBN 9780321356703. Available from: <https://books.google.com.br/books?id=HCQdypbpZXgC>.

MEINICKE, Jens et al. **Mastering Software Variability with FeatureIDE**. 1st: Springer Publishing Company, Incorporated, 2017. ISBN 3319614428.

MIANI, Rodrigo Sanches; ZARPELAO, Bruno Bogaz; MENDES, Leonardo de Souza. An Investigation About the Absence of Validation on Security Quantification Methods. **SBSI 2015 Proceedings of the annual conference on Brazilian Symposium on Information Systems**, May, p. 315–322, 2015. Available from: <http://dl.acm.org/citation.cfm?id=2814058.2814109%5Cnhttp://www.lbd.dcc.ufmg.br/colecoes/sbsi/2015/043.pdf>.

MICROSOFT. **The STRIDE Threat Model**. 2009. Available from: <https://docs.microsoft.com/en-us/previous-versions/commerce-server/ee823878(v=cs.20)?redirectedfrom=MSDN>.

MILLER, C.; VALASEK, C. **Remote exploitation of an unaltered passenger vehicle. Rockwell Collins Inc., Tech. Rep.** 2015. Available from: <http://illmatics.com/Remote%5C%20Car%5C%20Hacking.pdf>.

MONTECCHI, Leonardo; GALLINA, Barbara. SafeConcert: A metamodel for a concerted safety modeling of socio-technical systems. **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**, 10437 LNCS, p. 129–144, 2017. ISSN 16113349.

MONTECCHI, Leonardo; PURI, Stefano. **CHESS-SBA: State-Based Analysis**. 2020. Available from: <https://github.com/montex/CHESS-SBA>.

MOORE, Andrew; ELLISON, Robert; LINGER, Richard. **Attack Modeling for Information Security and Survivability**. Pittsburgh, PA, 2001. Available from: <http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=5417>.

MOTTAHIR, Md.; IRSHAD, Asif; ZAFAR, Aasim. A Comprehensive Study of Software Product Line Frameworks. **International Journal of Computer Applications**, v. 151, n. 3, 2016. DOI: 10.5120/ijca2016911698.

MUNK, Peter; NORDMANN, Arne. Model-based safety assessment with SysML and component fault trees: application and lessons learned. **Software and Systems Modeling**, Springer Berlin Heidelberg, 2020. ISSN 16191374.

NATIONAL SCIENCE FOUNDATION. **Cyber-Physical Systems (CPS) Important Information And Revision Notes**. 2012. Available from: <https://www.nsf.gov/pubs/2011/nsf11516/nsf11516.pdf>.

OLIVEIRA, Andre Luiz de; BRAGA, Rosana T. V.; MASIERO, Paulo C.; PAPADOPOULOS, Yiannis, et al. Variability Management in Safety-Critical Software Product Line Engineering. In: INTERNATIONAL Conference on Software Reuse - ICSR 2018. 2018.

OLIVEIRA, André Luiz de; BRAGA, Rosana; MASIERO, Paulo; PAPADOPOULOS, Yiannis, et al. Variability Management in Safety-Critical Software Product Line Engineering. In: INTERNATIONAL Conference on Software Reuse - ICSR 2018. Springer Verlag, 2018. 10826 LNCS, p. 3–22. ISBN 9783319904207. DOI: 10.1007/978-3-319-90421-4{_}1. Available from: <http://link.springer.com/10.1007/978-3-319-90421-4_1>.

OLIVEIRA, André Luiz de; BRAGA, Rosana, et al. Variability management in safety-critical systems design and dependability analysis. **Journal of Software: Evolution and Process**, v. 31, n. 8, e2202, Aug. 2019. ISSN 20477473. Available from: <`http://doi.wiley.com/10.1002/smr.2202`>.

OLIVEIRA, Edson A.; GIMENES, Itana M.S.; MALDONADO, José C. Systematic management of variability in UML-based software product lines. **Journal of Universal Computer Science**, v. 16, n. 17, p. 2374–2393, 2010. ISSN 0958695X.

OMG. **Meta-Object Facility**. 2019. Available from: <`http://https//www.omg.org/mof/,last`>.

OMG. **OMG Systems Modeling Language SysML**. 2017. Available from: <`https://www.omg.org/spec/SysML/1.6/PDF`>.

OMG. **OMG Systems Modeling Language SysML**. 2017. Available from: <`https://www.omg.org/spec/SysML/1.6/PDF`>.

OMG. **UML MARTE: Modeling and Analysis of Real-time and Embedded Systems**. 2019. `https://www.omg.org/omgmarte/`.

OMG. **Unified Modeling Language (UML) 2.5**. 2017. Available from: <`https://www.omg.org/spec/UML/2.5.1/PDF`>.

OMG. **Unified Modeling Language (UML) 2.5**. 2017. Available from: <`https://www.omg.org/spec/UML/2.5.1/PDF`>.

ORG., Python. **The Element Tree XML API**. 2020. https://docs.python.org/2/library/xml.etree.elementtree.html.

OSIS, Janis; DONINS, Uldis. **Topological UML Modeling: An Improved Approach for Domain Modeling and Software Development**. 2017. P. 133–151. DOI: `10.1016/b978-0-12-805476-5.00005-8`.

PAPADOPOULOS, Yiannis et al. Engineering failure analysis and design optimisation with HiP-HOPS. **Engineering Failure Analysis**, v. 18, n. 2, p. 590–608, 2011. ISSN 1350-6307.

POHL, Klaus; BÖCKLE, Günter; LINDEN, Frank J van der. **Software Product Line Engineering: Foundations, Principles, and Techniques**. Springer, 2005. ISBN 978-3-540-24372-4.

POHL, Richard; HÖCHSMANN, Mischa, et al. Variant Management Solution for Large Scale Software Product Lines. In: 2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP). 2018. P. 85–94.

PURE-SYSTEMS. **pure::variants**. 2021. Available from: <`https://www.pure-systems.com,%20last%20visited%202021-09-25`>.

PURE-SYSTEMS GMBH. **pure::variants User's Guide**. 2020. Available from: <https://www.pure-systems.com/mediapool/pv-user-manual.pdf>.

PYPARSING. **PyParsing Documentation**. 2020. https://pyparsing-docs.readthedocs.io/en/latest/.

RABINER, L.; JUANG, B. An introduction to hidden Markov models. **IEEE ASSP Magazine**, v. 3, n. 1, p. 4–16, 1986. DOI: 10.1109/MASSP.1986.1165342.

RETP. **Régie Autonome des Transports Parisiens**. 2021. Available from: <https://www.ratp.fr/en>.

REULING, Dennis et al. Towards projectional editing for model-based SPLs. In: 14TH International Working Conference on Variability Modelling of Software-Intensive Systems, VaMoS 2020. Magdeburg, Germany, 2020. ISBN 9781450375016. DOI: 10.1145/3377024.3377030.

ROBERTS, P. R. et al. A leadless pacemaker in the real-world setting: The Micra Transcatheter Pacing System Post-Approval Registry. **Heart rhythm**, v. 14, n. 9, p. 1375–1379, 2017. DOI: 10.1016/j.hrthm.2017.05.017.

RTCA. **DO-178C Software Considerations in Airborne Systems and Equipment Certification**. Radio Technical Commission for Aeronautics, 2011.

RTCA. **DO-356 Airworthiness Security Methods and Considerations**. 2014.

SAE. J3061. In: CYBERSECURITY Guidebook for Cyber-Physical Vehicle Systems. SAE International, 2016.

SAE. **SAE ARP 4754A: Guidelines for development of Civil Aircraft and Systems**. SAE International, 2010.

SAE INTERNATIONAL. **ARP4761 - Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment**. 1996.

SAE INTERNATIONAL. **SAE ARP 4754A - Guidelines for Development of Civil Aircraft and Systems**. 2010. Available from: <https://www.sae.org/standards/content/arp4754a/>.

SALTZER, J. H.; SCHROEDER, M. D. The protection of information in computer systems. **Proceedings of the IEEE**, v. 63, n. 9, p. 1278–1308, 1975. DOI: 10.1109/PROC.1975.9939.

SCHNEIDER, Daniel et al. WAP: Digital Dependability Identities. In: PROCEEDINGS of the 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE). USA: IEEE Computer Society, 2015. (ISSRE '15), p. 324–329. ISBN 9781509004065. DOI: 10.1109/ISSRE.2015.7381825. Available from: <https://doi.org/10.1109/ISSRE.2015.7381825>.

SCHULZE, Michael; MAUERSBERGER, Jan; BEUCHE, Danilo. Functional safety and variability: Can it be brought together? **ACM International Conference Proceeding Series**, p. 236–243, 2013.

SCHWINN, Jean-Pascal; ADLER, Rasmus; KEMMANN, Sören. Combining safety engineering and product line engineering. In: SOFTWARE Engineering 2013 - Workshopband. 2013. P. 545–554.

SOMMERVILLE, Ian. **Software Engineering**. 10th: Pearson, 2015. ISBN 0133943038, 9780133943030.

SPLC HALL OF THE FAME. **SPLC hall of the fame: General Motors Powertrain**. 2019. Available from: <`https://splc.net/fame/general-motors-powertrain`>.

SRIVATANAKUL, Thitima; CLARK, John A.; POLACK, Fiona. Effective Security Requirements Analysis: HAZOP and Use Cases. In: 2004. P. 416–427. Available from: <`http://link.springer.com/10.1007/978-3-540-30144-8_35`>.

STEINER, Eduardo; MASIERO, Paulo; BONIFÁCIO, Rodrigo. Managing SPL Variabilities in UAV Simulink Models with Pure::variants and Hephaestus. **CLEI Electronic Journal**, v. 16, n. 1, p. 1–16, 2013. ISSN 0717-5000.

STEWART, D. et al. Architectural Modeling and Analysis for Safety Engineering. In: BOZZANO, Marco; PAPADOPOULOS, Yiannis (Eds.). **Model-Based Safety and Assessment**. Cham: Springer International Publishing, 2017. P. 97–111. ISBN 978-3-319-64119-5.

STOREY, Neil R. **Safety Critical Computer Systems**. USA: Addison-Wesley Longman Publishing Co., Inc., 1996. ISBN 0201427877.

SWIDERSKI, F.; SNYDER, W. **Threat Modeling**. USA: Microsoft Press, 2004. ISBN 0735619913.

TISCHER, Christian et al. Experiences from a Large Scale Software Product Line Merger in the Automotive Domain. In: (SPLC '11). ISBN 9780769544878. DOI: `10.1109/SPLC.2011.15`. Available from: <`https://doi.org/10.1109/SPLC.2011.15`>.

US MILITARY. **Procedure for Performing a Failure Mode Effect and Criticality Analysis**. 1977.

VARA, Jose Luis de la et al. AMASS: A Large-Scale European Project to Improve the Assurance and Certification of Cyber-Physical Systems. **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**, 11915 LNCS, p. 626–632, 2019. ISSN 16113349.

VASILEVSKIY, Anatoly et al. The BVR tool bundle to support product line engineering. In: PROCEEDINGS of the 19th International Conference on Software Product Line - SPLC '15. New York, New York, USA: ACM Press, 2015. v. 8769, p. 380–384. ISBN 9781450336130.

VESELY, W. E. et al. **Fault tree handbook with aerospace applications**. 2002.

WALKER, Martin et al. Automatic optimisation of system architectures using EAST-ADL. **Journal of Systems and Software**, v. 86, n. 10, p. 2467–2487, Oct. 2013. ISSN 01641212. DOI: 10.1016/j.jss.2013.04.001.

WALLACE, Malcolm. Modular architectural representation and analysis of fault propagation and transformation. **Electronic Notes in Theoretical Computer Science**, v. 141, n. 3, p. 53–71, 2005. ISSN 15710661. DOI: 10.1016/j.entcs.2005.02.051.

WÖLFL, Andreas et al. Generating qualifiable avionics software: An experience report. In: PROCEEDINGS - 2015 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015. 2016. P. 726–736. ISBN 9781509000241.

WÖLFL, Andreas et al. Generating Qualifiable Avionics Software: An Experience Report (E). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2015. P. 726–736. DOI: 10.1109/ASE.2015.35.

WOLSCHKE, Christian et al. Industrial Perspective on Reuse of Safety Artifacts in Software Product Lines. In: PROCEEDINGS of the 23rd International Systems and Software Product Line Conference - Volume A. New York, NY, USA: ACM, Sept. 2019. P. 143–154. ISBN 9781450371384. DOI: 10.1145/3336294.3336315. Available from: <https://dl.acm.org/doi/10.1145/3336294.3336315>.

YOUNG, Bobbi; CLEMENTS, Paul. Model Based Engineering and Product Line Engineering: Combining Two Powerful Approaches at Raytheon. In: 27TH Annual INCOSE International Symposium. 2017. DOI: 10.1002/inst.12248.

ZIADI, Tewfik; HÉLOUËT, Loïc; JÉZÉQUEL, Jean-Marc. Towards a UML Profile for Software Product Lines. In: INTERNATIONAL Workshop on Software Product-Family Engineering. 2004. P. 129–139. DOI: 10.1007/978-3-540-24667-1{_}10. Available from: <http://link.springer.com/10.1007/978-3-540-24667-1_10>.