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

Luis Felipe de Almeida Nascimento

A Model-Driven Methodology to Support Runtime Assurance of Open and Adaptive Systems

Juiz de Fora

2024

Luis Felipe de Almeida Nascimento

A Model-Driven Methodology to Support Runtime Assurance of Open and Adaptive 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: Prof. Dra. Regina Maria Maciel Braga Villela

Juiz de Fora

2024

Ficha catalográfica elaborada através do Modelo Latex do CDC da UFJF
com os dados fornecidos pelo(a) autor(a)

Nascimento, Luis Felipe de Almeida.
A Model-Driven Methodology to Support Runtime Assurance of Open and Adaptive Systems / Luis Felipe de Almeida Nascimento. – 2024.
118 f. : il.

Orientador: André Luiz de Oliveira
Coorientador: Regina Maria Maciel Braga Villela
Dissertação (Mestrado) – Universidade Federal de Juiz de Fora, Instituto de Ciências Exatas. Programa de Pós Graduação em Ciência da Computação, 2024.

1. Cyber-Physical Systems. 2. Assurance Cases. 3. SACM. I. de Oliveira, André Luiz, orient. II. Villela, Regina Maria Maciel Braga, coorient. III. Título

Luis Felipe de Almeida Nascimento

A model-driven methodology to support runtime assurance of open and adaptive 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: Ciência da Computação.

Aprovada em 23 de setembro de 2024.

BANCA EXAMINADORA

Prof. Dr. André Luiz de Oliveira - Orientador
Universidade Federal de Juiz de Fora

Profª. Dra. Regina Maria Maciel Braga Villela - Coorientadora
Universidade Federal de Juiz de Fora

Profª. Dra. Fernanda Cláudia Alves Campos
Universidade Federal de Juiz de Fora

Prof. Dr. Leonardo Montecchi
Norwegian University of Science and Technology

Profª. Dra. Rosana Teresinha Vaccare Braga

Universidade de São Paulo

Juiz de Fora, 10/09/2024.

Documento assinado eletronicamente por **Rosana Teresinha Vaccare Braga**, **Usuário Externo**, em 23/09/2024, às 11:40, 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 23/09/2024, às 12:19, 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 **Leonardo Montecchi**, **Usuário Externo**, em 24/09/2024, às 18:51, 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 **Andre Luiz de Oliveira**, **Professor(a)**, em 24/09/2024, às 19:45, 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 26/09/2024, às 07:29, 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 **1977522** e o código CRC **B2F79894**.

RESUMO

Contexto: Sistemas ciber-físicos (CPSs) abrigam vasto potencial econômico e impacto social, permitindo novos tipos de aplicações promissoras em diferentes domínios de sistemas embarcados como automotivo, aviônicos, ferroviários, de saúde e automação. Esses sistemas desempenham funções críticas e, se falharem, podem prejudicar pessoas ou levar ao colapso de infraestruturas importantes com consequências catastróficas para a indústria e/ou sociedade. Dessa forma, o desenvolvimento de CPSs demanda a demonstração de propriedades de segurança, proteção e confiabilidade. Casos de garantia fornecem um meio explícito para justificar/avaliar as propriedades de confiança do sistema com referências explícitas e implícitas a artefatos de design, de análise segurança e de confiabilidade. O Structured Assurance Case Metamodel (SACM) define um metamodelo padronizado para representar casos de garantia estruturados. O standard SACM fornece a base para a garantia de segurança de sistema dirigida por modelos com grande potencial para ser aplicada em domínios de CPSs abertos e adaptativos emergentes. **Motivação:** A natureza aberta (capacidade de se conectar) e adaptativa (adaptar a mudanças de contextos) dos CPSs exige uma mudança de paradigma de garantia de segurança do sistema em tempo de design para a garantia em tempo de execução. Assim, espera-se que os casos de garantia sejam intercambeados, integrados e verificados em tempo de execução para garantir a confiança de CPSs. Entretando, possibilitar a rastreabilidade entre os casos de garantia e modelos de design, análise e de processso do sistema, que são parte da Identidade de Confiança Digital Executável (EDDIs) de componentes de um CPSs, ainda é uma barreira. **Objetivo:** Neste trabalho é proposta uma nova metodologia dirigida por modelos para apoiar a especificação e a síntese de casos de garantia executáveis a partir de um conjunto de modelos de sistema para demonstrar a segurança e proteção de CPSs em tempo de execução. Este trabalho aprimora a especificação SACM com *pattern extensions* que adicionam semântica ao conceito de *ImplementationConstraint* para possibilitar a especificação de padrões de caso de garantia executáveis (*templates*) vinculados a informações de modelos de projeto, análise e de processo (evidências) que constituem a EDDI de um CPS e/ou de um componente do CPS. A metodologia proposta inclui um método de apoio à especificação e à instanciação de padrões de caso de garantia, uma ferramenta de modelagem com a capacidade de especificar padrões de caso de garantia executáveis vinculados a informações de uma EDDI e um algoritmo de instanciação para apoiar a síntese automática de casos de garantia. **Avaliação:** A viabilidade da metodologia proposta foi avaliada por meio de sua aplicação na geração de casos de garantia para dois sistemas do domínio automotivo. **Resultados:** A efetividade da metodologia proposta foi demostrada pelo baixo tempo de excução da instanciação, alta precisão e omissões mínimas na geração de casos de garantia para dois sistemas de tamanho médio.
Palavras-chave: Sistemas Ciber-Físicos. Casos de Garantia. SACM.

ABSTRACT

Context: Cyber-physical systems (CPS) harbor vast economic potential and societal impact, enabling new types of promising applications in different embedded system domains such as automotive, avionics, railway, healthcare, and home automation. These systems perform safety-critical functions, thus in case of failure, they may harm people or lead to the collapse of important infrastructures with catastrophic consequences to industry and/or society. Therefore, CPSs demand the justification of system safety and component reliability. Assurance cases provide an explicit means for justifying/assessing confidence in system dependability with explicit and implicit references to design, safety, and reliability artifacts. The Structured Assurance Case Metamodel (SACM) defines a standardized metamodel for representing structured assurance cases. SACM provides the foundations for model-based system assurance with potential to be applied in emergent open and adaptive CPS domains. **Motivation**: CPSs are loosely connected and come together as temporary configurations of smaller systems that may dissolve and give place to other configurations. The open and adaptive nature of CPSs, demands a paradigm shift from design-time to runtime system assurance. To ensure the dependability of CPSs, assurance cases are expected to be exchanged, integrated, and verified at runtime. However, enabling the traceability between assurance cases and system design, analysis, and process models, which are part of Executable Digital Dependability Identities (EDDIs) of CPS components, is still a barrier. **Objective**: This study introduces a novel model-driven methodology to support the specification and synthesis of executable SACM assurance cases from various systems models to demonstrate CPS safety and security at runtime. This work also enhances the SACM standard with pattern extensions that added semantics to the concept of Implementation Constraint to support the specification of executable assurance case pattern (templates) specifications linked to information from design, analysis, and process models (evidence) that constitute the EDDI of a CPS and/or CPS component. This study also provides an assurance case pattern specification and instantiation methodology, comprising a modeling tool for specifying executable assurance case patterns linked to EDDI information, and an instantiation algorithm to support the automatic synthesis of assurance cases from EDDI information. **Evaluation**: The feasibility of the proposed methodology is evaluated in two illustrative studies from the automotive domain. **Results**: The effectiveness of the proposed methodology are demonstrated by low instantiation execution time, higher accuracy, and minimal omissions in the generated product assurance cases for the two medium-sized automotive systems.
Keywords: Cyber-Physical Systems. Assurance Cases. SACM.

FIGURES

TABLES

SUMMARY

1 INTRODUCTION

1.1 CONTEXT

Cyber-physical systems (CPS) integrate physical processes and computer systems that contain sensors to observe the environment and actuators to influence physical processes (KOPETZ et al., 2016). CPSs harbor vast economic potential and societal impact, enabling new types of promising applications in different embedded system domains such as automotive[1] (DAJSUREN; BRAND, 2019), avionics[2] (ASLANSEFAT et al., 2022), railway, healthcare, and home automation (WEI, R.; KELLY, T. P.; HAWKINS, R., et al., 2017). These systems perform safety-critical functions, thus in case of failure, they may harm people or lead to the collapse of important infrastructures with catastrophic consequences to industry and/or society (TRAPP; SCHNEIDER; LIGGESMEYER, 2013).

Safety-critical CPS demand justification of system safety and component reliability. Assurance cases provide an explicit means for justifying and assessing confidence in safety, security, and other dependability properties of interest. An assurance case is a structured argument formed by a set of claims arguing and justifying the assurance of system safety and security properties in a particular operating environment based on compelling evidence (e.g., design, analysis, and process models) and the relationships between these elements (HAWKINS; KELLY, 2010). An assurance case supported by evidence is the key artifact for the safety/security acceptance of systems before their release for operation. The required system assurance evidence can be gathered by applying safety and reliability analysis techniques. Fault Tree Analysis (FTA) (NASA, 2002) and Failure Modes and Effects Analysis (FMEA) are among the most popular safety analysis techniques. These techniques support engineers in determining how a system can fail and the likelihood of those failures. FTA and FMEA constitute integral parts of Model-Based Safety Analysis (MBSA).

1.2 MOTIVATION

CPSs are loosely connected and come together as temporary configurations of smaller systems that may dissolve and give place to other configurations. Given the highly open and adaptive nature of CPSs, i.e., systems can connect at runtime and adapt their behaviors to changing contexts, the evidence and justification of system assurance need to change from design time to runtime. With this shift, there will be a transition from the current design time assurance cases produced from manually created artifacts to assurance case models that can be automatically synthesized and evaluated at runtime (WEI; KELLY, et al., 2018).

[1] Autonomous cars
[2] Unmanned aerial vehicles

To achieve this goal, it is necessary to equip a CPS or a CPS component with all the information that uniquely describes its dependability characteristics (design, analysis, and process models). This collection of models and dependability information constitutes a system Executable Digital Dependability Identity (EDDI) (WEI, Ran; KELLY, Tim P; HAWKINS, Richard, et al., 2018). EDDIs contain all the required information for system assurance, when produced at design time provide the basis for automated integration of components into systems at development-time, and dynamic integration of independent systems into systems of systems at runtime. To ensure the dependability of CPSs, assurance cases are expected to be generated, exchanged, integrated, and verified at runtime (WEI, Ran; KELLY, Tim P; DAI, Xiaotian, et al., 2019). The Structured Assurance Case Metamodel (SACM) (OMG, 2021), issued by the Object Management Group (OMG), provides the foundations for runtime model-based system assurance. SACM defines a standardized metamodel and visual notation for representing structured assurance cases. SACM was developed to support interoperability between different assurance case approaches, e.g., GSN (GSN, 2018), CAE (BLOOMFIELD, Robin; BISHOP, Peter, 2009). The SACM was used in the DEIS project (DEIS, 2020) as a backbone for its Open Dependability Exchange (ODE) metamodel, which defines the appropriate format of an EDDI, in the first step towards runtime assurance of CPSs. SACM enables the specification, traceability to evidence, and automated synthesis of executable SACM assurance case patterns (*templates*) from design and analysis (ODE) models. Assurance case patterns are useful in capturing good practice in system argumentation for re-use, by defining the required system information to instantiate abstract assurance claims and evidence to support those claims (KELLY; MCDERMID, 1997). Although SACM and ODE metamodels provide the backbone for the assurance of CPSs at runtime, it is still a barrier enabling the traceability between the assurance case and system design, analysis, and process models which are part of the EDDI of open and adaptive CPS components, at runtime.

1.3 OBJECTIVE

This study introduces a novel model-driven methodology to support the specification and synthesis of executable assurance cases from various system models. The primary goal is to enable the demonstration of the safety and security of Cyber-Physical Systems (CPS) at runtime. The first contribution of this work is the enhancement of the SACM standard with pattern extensions. These extensions add semantics to the concept of SACM Implementation Constraint, which is crucial to support the specification of executable assurance case patterns (templates). These extensions support the specification of traceability links between abstract elements of assurance case patterns and information from design, safety/security analysis, and process models, i.e., the system assurance evidence. Such evidence constitutes the Executable Digital Dependability Identity (EDDI) of a CPS

and/or CPS component.

In addition, this study presents a methodology to support the specification and instantiation of executable assurance case patterns. The proposed methodology is supported by a modeling tool for specifying executable assurance case patterns linked to EDDI information. Moreover, an instantiation algorithm was developed to enable the synthesis of executable assurance case patterns from EDDI information. This algorithm ensures that the SACM assurance case patterns are executable, and accurately reflect the safety and security properties of the CPS stored in its EDDI. The feasibility of the proposed methodology was evaluated in two medium-sized CPSs from the automotive domain: the Hybrid Braking System (HBS) and the Highly Automated Driving (HAD) system. The results demonstrated the effectiveness of the proposed methodology in generating assurance cases that correspond to the expected output for each one of those case studies. The results also demonstrate the efficiency of the developed algorithm concerning the execution time from the order milliseconds taken to synthesize assurance cases for both systems. Figure 1.1 provides an overview of the contributions of this work and their interconnections. The SACM pattern extensions provide conceptual definitions for the methodology and the semantics for the instantiation algorithm. The SACM pattern extensions are supported by the SACM ACEditor. The proposed methodology outlines a systematic method to specify and instantiate SACM assurance case patterns. Lastly, the instantiation algorithm is implemented by the SACM ACEditor to automatically instantiate assurance case patterns.

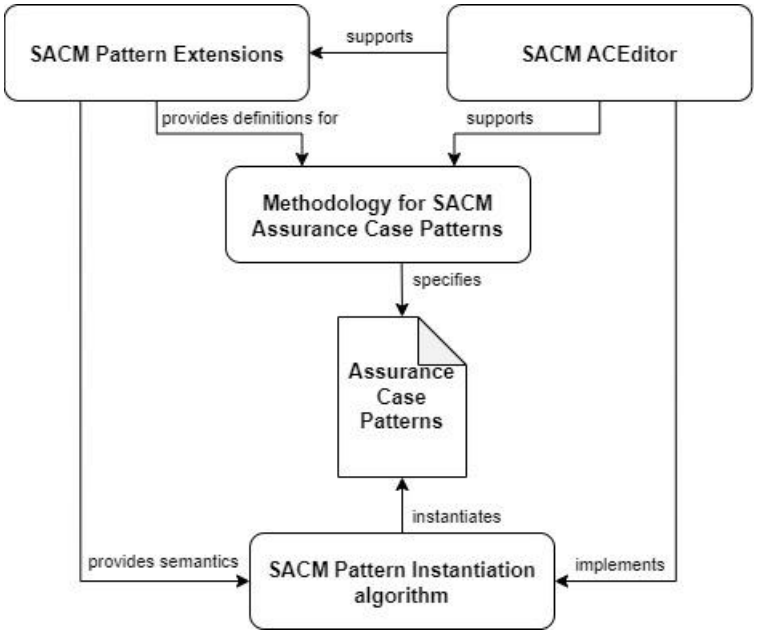

Figure 1.1 – Contributions Overview.

1.4 ORGANIZATION

This dissertation is organized into eight chapters. **Chapter 2** presents the basic concepts, e.g., cyber-physical system, safety terminology, assurance cases, Digital Dependability Identity, Model-Driven Engineering, SACM, and the contextualization of related work, needed for the reader to understand the contributions of this research. **Chapter 3** introduces the SACM patterns extensions for specifying executable assurance case patterns. **Chapter 4** describes the Assurance Case Editor (ACEditor) modeling tool to support the specification of executable assurance case patterns linked to the EDDI information. **Chapter 5** presents a Model-Driven methodology to enable the runtime assurance of open and adaptive systems. **Chapter 6** presents a instantiation algorithm, developed to support the automatic instantiation of executable assurance case patterns from EDDI information. **Chapter 7** describes the feasibility evaluation of the proposed methodology in two medium-sized CPSs models from the automotive domain. **Chapter 8** highlights the contributions, limitations of this study, and future research directions.

2 BACKGROUND

This chapter presents the background concepts needed for the reader to understand the context of the research contributions. Section 2.1 describes Cyber-Physical Systems and their properties. Section 2.2 explores some of the terminology used in this work. Section 2.3 presents an overview of Safety life-cycle and safety standards. Section 2.4 describes some of the main safety analysis techniques used for system assurance. Section 2.5 describes the Model-Driven Engineering (MDE) and Section 2.6 describes the Model-Based Safety Analysis (MBSA). Section 2.7 explores GSN-compliant assurance cases. The Structured Assurance Case Metamodel (SACM) is detailed in Section 2.8. Section 2.9 explores the concept of Runtime Assurance. Section 2.10 describes the concepts of system Digital Dependability Identities (DDIs) and Section 2.11 describes the Open-Dependability Exchange metamodel (ODE). Finally, Section 2.12 explores the related works.

2.1 CYBER-PHYSICAL SYSTEMS (CPS)

Cyber-Physical Systems (CPS) integrate physical processes and computer systems that contain sensors to observe the environment and actuators to influence physical processes (KOPETZ et al., 2016). CPSs harbor vast economic potential and societal impact, enabling new types of promising applications in different embedded system domains such as automotive[1] (DAJSUREN; BRAND, 2019), avionics[2] (ASLANSEFAT et al., 2022), railway, healthcare, and home automation (WEI, R.; KELLY, T. P.; HAWKINS, R., et al., 2017). CPSs in such domains are highly open (systems can connect at runtime), and adaptive (they are capable of adapting to changing contexts). Figure 2.1 illustrates a typical example of a CPS, where sensors capture information about the physical process sending it for computation through a network and resulting in actions from the actuators. These systems perform safety-critical functions and if they fail may harm people or lead to the collapse of important infrastructures with catastrophic consequences to industry, and/or society (TRAPP; SCHNEIDER; LIGGESMEYER, 2013). CPSs are loosely connected and come together as temporary configurations of smaller systems that may dissolve and give place to other configurations. The key problem in assessing the safety and security of CPS is that it is almost impossible to anticipate the concrete CPS structure (configuration), capabilities, and environment at design time. Thus, since most open and adaptive CPSs are safety-critical, it is imperative to assure the safety, security, and other dependability properties of a CPS/CPS not only at design time but also at runtime (WEI, R.; KELLY, T. P.; DAI, X., et al., 2019; WEI; KELLY, et al., 2018).

[1] Autonomous cars
[2] Unmanned aerial vehicles

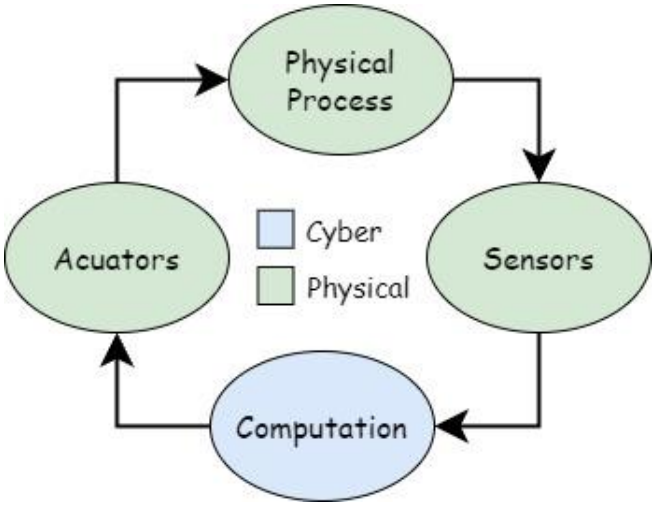

Figure 2.1 – General example of CPS.

Safety standards establish guidelines to ensure the dependability properties of CPS, e.g., ISO 26262 for the automotive domain and SAE ARP 4754A aircraft domain. These guidelines must be followed in the development and safety life cycle. Safety standards require the analysis and demonstration of the safety properties of a critical at different levels of abstraction, from requirements to component implementation. When a critical system addresses all the requirements posed by safety standards, it is qualified to receive the certification and release for operation.

2.2 TERMINOLOGY

Dependability is the ability of a system to deliver service that can justifiably be trusted. This integrative concept encompasses availability, reliability, safety, confidentiality, integrity, maintainability, and security. **Availability** is readiness for correct service. **Reliability** is the continuity of correct service. **Safety** is the absence of catastrophic consequences on the user(s) and the environment. **Confidentiality** is the absence of unauthorized disclosure of information. **Integrity** is the absence of improper system state alterations. **Maintainability** is the ability to undergo repairs and modifications. **Security** is the concurrent existence of availability for authorized users only with confidentiality and integrity (AVIZIENIS; LAPRIE; RANDELL, et al., 2001).

Hazard is defined as the potential source of harm caused by the malfunctioning behavior of a system or component. **Risk** is the combination of the likelihood of harm (i.e., death, physical injury, and damage to people, property, or environment) and the severity of that harm. **Failure** is the inability of a system or system component to perform a required function within specified limits when a fault is encountered. **Fault** is the manifestation of an error, which could lead to a failure. **Safety requirement** is the required risk reduction measures associated with a given hazard or component failure. **Assurance** is the planned and systematic actions necessary to provide adequate confidence and evidence that a

system satisfies the given safety requirements. **Evidence** is the information that serves as the grounds and starting-point of (safety) arguments, based on which the degree of truth of the claims in arguments can be established, challenged, and contextualized (OLIVEIRA, 2016).

2.3 SAFETY LIFE-CYCLE

Standards define the concept of Safety Integrity Levels (SILs) to allocate requirements to mitigate hazard, subsystem, and/or component failure effects on the overall system safety. The IEC 61508 (BROWN, 2000) industry standard defines four different safety integrity levels, ranging from the least stringent SIL 1 to the most stringent SIL 4. The most stringent SILs demand significant verification and validation effort such as performing control-flow and data-flow testing to be achieved. On the other hand, achieving the least stringent SILs demands less rigorous verification activities such as software inspection. Standards also define a risk-based approach, e.g., based on severity and likelihood levels, to classify the risk posed by system hazards and component failures in a given SIL, and they prescribe requirements and provide guidance to achieve safety per SIL. It also defines rules for decomposing safety requirements allocated to mitigate hazard effects (in the form of SILs) into requirements to mitigate architectural subsystems and component failures.

In the automotive domain, the ISO 26262 (ISO, 2018) standard defines the requirements for ensuring functional safety in electrical/electronic systems of small and medium-sized general-purpose road vehicles (Figure 2.2). This standard introduces a comprehensive safety lifecycle, a risk-based approach to determine Automotive Safety Integrity Levels (ASILs), requirements for validation, and confirmation measures to ensure an acceptable level of safety. At the concept phase (ISO 26262 Part 3), firstly, functions are allocated to systems, subsystems, and components, which can be software, electric, or electronic at the item definition (Sec. 3.5) of Figure 2.2. Then, item dependencies and their interactions with the environment at the vehicle level are specified to initiate the safety lifecycle (Sec. 3.6). After the initiation of the safety lifecycle, Hazard Analysis and Risk Assessment (HARA) is performed to identify and categorize the hazards that malfunction the item (Sec. 3.7) at the system level, e.g., using HAZard and OPerability studies. Safety goals, in the form of ASILs, are then allocated to eliminate or minimize hazard effects, and decomposed into functional and technical safety requirements, at the Functional Safety Concept (Sec. 3.8), e.g., using Fault Tree Analysis (FTA) and Failure Modes and Effects Analysis (FMEA) safety analysis techniques. System safety requirements, e.g., fault prevention, detection and removal, tolerance, and/or forecasting, assigned to mitigate hazard effects are preliminary allocated to architectural elements (i.e., Functional Safety Requirements - FSRs) of the item to mitigate hazardous behaviors

into subsystems. FSRs allocated to architectural subsystems are finally decomposed into technical safety requirements (TSR) to mitigate component failure effects on safety. This standard also recommends the specification of an Assurance Case.

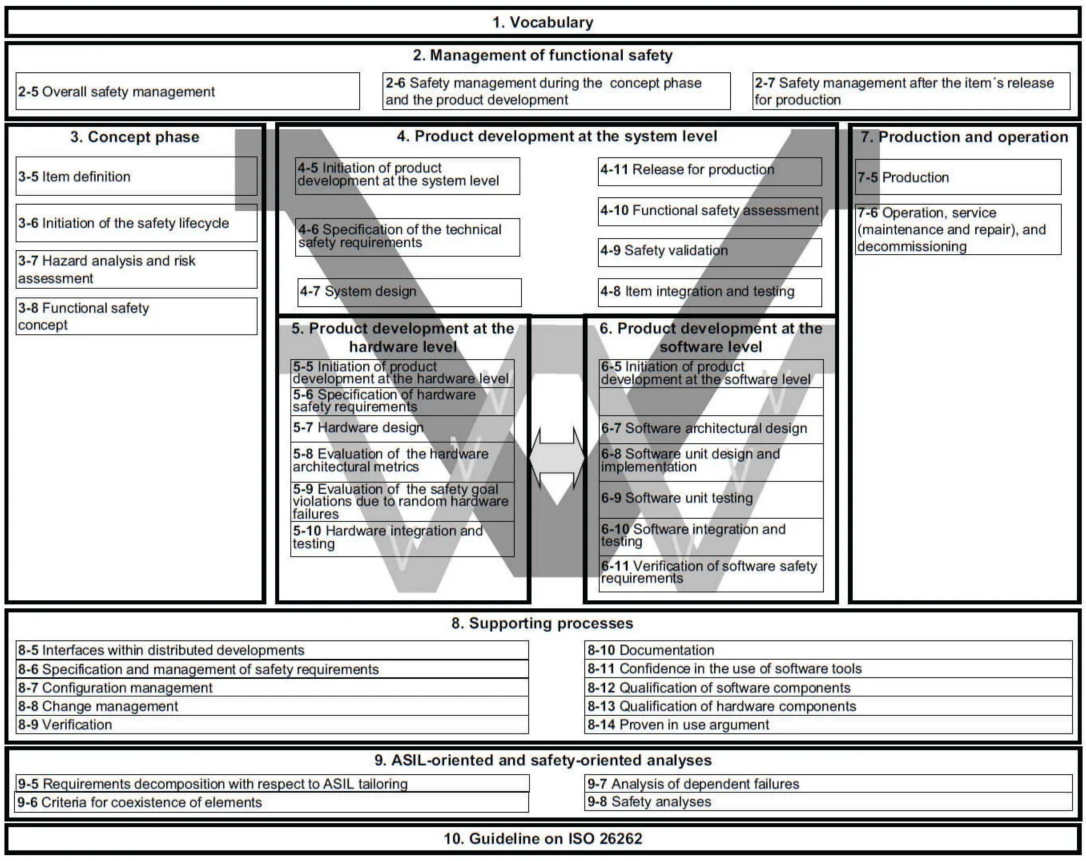

Figure 2.2 – ISO 26262 (ISO, 2018).

2.4 SAFETY ANALYSIS TECHNIQUES

This section presents two of the most popular and used safety analysis techniques and the concept of assurance case specification which is also relevant for understanding this work. Section **2.4.1** describes the Functional Hazard Analysis (FHA) technique. Section **2.4.2** describes the Fault Tree Analysis (FTA) technique. Section **2.4.3** presents the assurance case specification process.

2.4.1 Functional Hazard Analysis

Functional Hazard Analysis (FHA) is conducted at the beginning of the system life cycle, (JOSHI et al., 2006). This technique identifies and classifies the failure conditions at the appropriate level, considering both loss of function and malfunctions, associated with the target system functions and combinations of functions. The FHA establishes derived safety requirements needed to limit failure effects, e.g., design constraints, and

annunciation of failure conditions. Firstly high-level functions of the system and the failure conditions associated with these functions are considered. Then the effects of these failure conditions are determined and classified. These failure conditions can be further broken down through FHAs and Fault Trees. The failure conditions associated with each hazard are defined together with their respective safety goals and the proposed means for demonstrating compliance, (JOSHI et al., 2006).

Ref.	Functional Failure (Hazard)	Critical Operational Phase	Aircraft Manifestation	Criticality	Comment
1.1.1	Loss of Guidance Values	Approach	Presence of No Computed Data (NCD) should signal FD and AP disconnect.	Minor	Becomes major hazard, equivalent to incorrect guidance, if disconnect fails.
1.1.2	Incorrect Guidance Values	Approach	Gradual departure from references until detected by flight crew during check of primary flight data resulting in manual disconnect and manual flying.	Major	No difference to the AP between loss of guidance and incorrect guidance.

Figure 2.3 – Functional Hazard Analysis Example (TRIBBLE; LEMPIA; MILLER, 2002)

Example: Figure 2.4 shows an excerpt of the FHA for an aircraft system (TRIBBLE; LEMPIA; MILLER, 2002). The hazard "Loss of guidance values" is minor in this figure because the AP and FD components should sense the absence of data and disconnect. "Incorrect guidance values" hazard has a major criticality because it may cause the aircraft to drift from its intended flight plan producing a “significant reduction in safety margin”. This could also require a “significant increase in workload” on the part of the flight crew to return the aircraft to its intended flight plan.

2.4.2 Fault Tree Analysis

Fault Tree Analysis (FTA) is a widely used, top-down reasoning technique for safety and reliability analysis (NASA, 2002). FTA is based on a visual notation to support the analysis of combinations of fault propagation paths (i.e., using AND, OR, NOT gates), comprising intermediate and basic events, that can lead the system to fail (top event), causing harm to people, environment, or property. The starting point of this analysis is a given top event (i.e., a system failure). Then, the possible combinations of causes that could trigger the top event (system failure) are progressively explored until individual component failures (basic events) have been reached. The FTA supports qualitative (logical) and quantitative (probabilistic) analysis. FTA requires extensions to model dynamic relationships between causes and component failures (i.e., modeling the dynamic failure behavior). Fault trees have been used for diagnosis purposes in runtime applications. In the work of Aslansefat (ASLANSEFAT et al., 2022), sensor readings are connected to "complex basic events" (CBE) as inputs to the fault tree analysis. The failure model is

updated in real time to provide diagnoses, predictions of failures, and reliability evaluation at runtime. Runtime FTA is the first step towards the integration of fault trees into the system Executable Digital Dependability Identity.

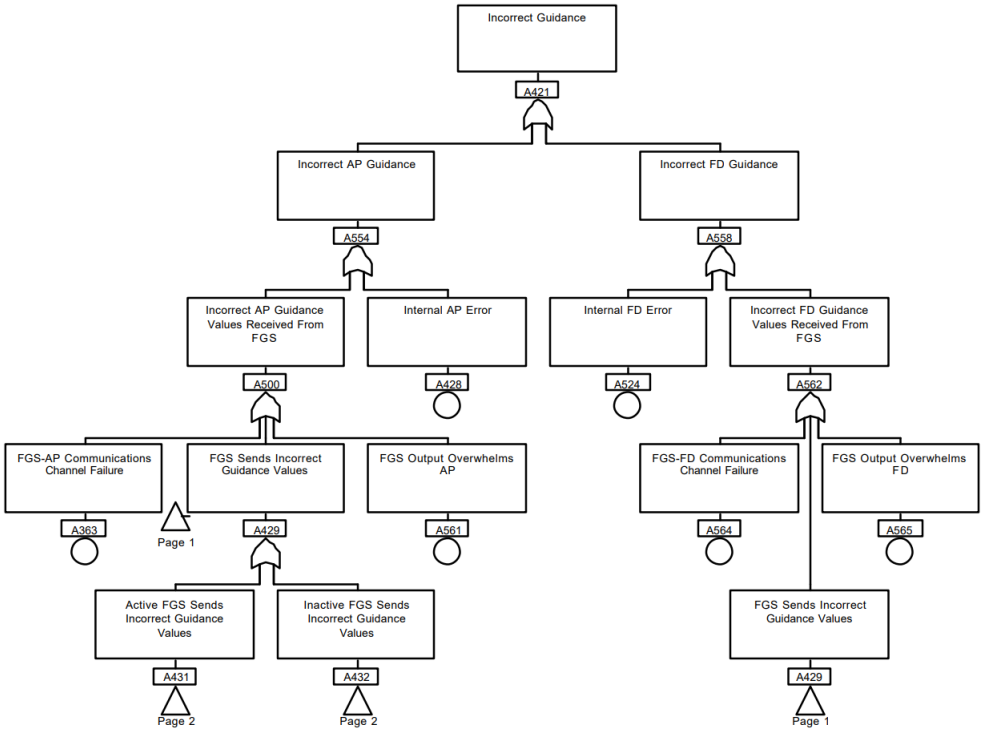

Figure 2.4 – FTA Excerpt for the Hazard – Incorrect Guidance (TRIBBLE; LEMPIA; MILLER, 2002).

Example: The top levels of the FTA for a hazard called Incorrect Guidance are shown in Figure 2.4. The fault tree root node first splits into two nodes “Incorrect AP Guidance” and “Incorrect FD Guidance” through an OR gate because the AP and the FD both receive guidance values from the FGS. At the next level, the “Internal Error” basic event addresses the possibility of AP, or FD corrupting correct data values provided to them by the FGS component. The OR gate “Incorrect Guidance Values Received From FGS” addresses the possibility that the FGS may pass incorrect values. These incorrect guidance values may, in turn, be due to the basic event “Communications Channel” or the gate “Output Overwhelms” hardware failures, in addition to the FGS internal event, “FGS Sends Incorrect Guidance Values” (TRIBBLE; LEMPIA; MILLER, 2002).

2.4.3 Assurance Case Specification

An Assurance Case or Safety Case is a **clear**, comprehensive and defensible **argument** that a **system** is **acceptably** safe to operate in a particular **context**, (KELLY; WEAVER, 2004). An assurance is **clear** because it communicates with a third party, it is an **argument** because must demonstrate how a reader can reach a reasonable conclusion about whether a **system** that is not limited to the conventional engineering "design" is

acceptably safe to operate in a particular context, because, a system be safe is considered an unobtainable goal. Thus, an assurance case has to be convincing about the system being safe enough to operate within a particular **context**. An assurance case should define the context because no system can be considered safe if it is used inappropriately or unexpectedly (KELLY; WEAVER, 2004).

The informal reuse of an assurance case is already commonplace, especially within stable and well-understood domains, e.g., aerospace engine controllers. However, informal reuse can fail, and in some cases be potentially dangerous. Several potential problems can arise where people are the main operators of cross-project reuse of assurance case artifacts and some of them are: artifacts being reused inappropriately; Reuse occurring in an ad-hoc fashion; Loss of knowledge; Lack of Consistency and Process Maturity; and Lack of traceability. To solve or minimize these problems, the reuse of an Assurance Case must be explicitly recognized and documented. This involves identifying and abstracting the reusable elements. Reuse of a specific assurance case, e.g., a particular fragment of evidence, can be highly unsuccessful because the structure of an assurance case may change from one system to another. However, the reuse of the general principles of a safety case is more successful than specific ones. General argumentation principles are present in different assurance cases. Therefore, an assurance case pattern describes a partial solution for arguing the system safety reusing the general structured (KELLY; MCDERMID, 1997). An Assurance Case pattern is a partial solution that references reusable elements and information, e.g., evidence, solutions, or artifacts required for the construction of systems safety arguments. The abstraction of details of a safety argument into a pattern is named abstract *term*, i.e., a reference to the required information. The information that will be used to instantiate a *term*, transforming it from abstract to non-abstract representation should be documented in the Assurance Case pattern specification.

Several notations have been proposed in the literature to document Assurance Cases and Assurance Case Patterns. Initially, a free text notation has been created. However, the ambiguity inherent to the natural language makes it difficult to accurately express complex arguments. Tabular notations have been created to overcome the limitations of natural language in expressing an assurance case. However, tabular notations have limitations in representing an argument whose assumptions and evidence are supported by another argument simultaneously (KELLY; WEAVER, 2004). The example below, extracted from a paper, illustrates the limitations of natural language assurance cases in clearly expressing complex arguments. In this example, references to sections of the textual assurance case are included in the text as needed, making it difficult to understand and discern the reasoning behind the argumentation.

> "For hazards associated with warnings, the assumptions of [7] Section 3.4 associated with the requirement to present a warning when no equipment

> failure has occurred are carried forward. In particular, with respect to hazard 17 in section 5.7 [4] that for test operation, operating limits will need to be introduced to protect against the hazard, whilst further data is gathered to determine the extent of the problem." (KELLY; WEAVER, 2004).

To overcome the lack of expressiveness from textual and tabular notations in representing complex argument structures, graphical notations have been created applying the Model-Driven Engineering concepts, (SELIC, 2003). Although graphical notations are not perfect, because each notation has its limitations, they are better than natural languages for representing an Assurance Case. Goal Structured Notation (GSN) (GSN, 2018) and the Structured Assurance Case Metamodel (SACM) (OMG, 2021) are examples of graphical notations that support the specification of assurance cases. These notations are discussed in Section 2.7 following the explanation of Model-Driven Engineering.

2.5 MODEL DRIVEN ENGINEERING

Model-Driven Engineering (MDE) focuses on models as the first-class entity of a software development process rather than computer programs. By creating abstract representations of systems, MDE helps streamline the design and implementation processes, allowing a more efficient and error-resistant software creation. A model is an abstraction with an intended and defined purpose (SELIC, 2003). An MDE approach includes Domain-Specific Modeling and Model Management. **Domain-specific Modeling** allows domain experts to capture the modeling concepts of their system in a meta-model, which aims to support the creation of system models according to the syntax and semantics of the language defined in the meta-model. A meta-model defines abstractions and rules to build computer readable specific models in a domain of interest, which establishes: an *abstract syntax* i.e. "the concepts from which models are created"; a *concrete syntax* i.e. "how rendering these concepts"; *well-formed rules* i.e. "rules for the application of the defined modeling concepts"; and the description of the semantics of a specific model (SELIC, 2003). **The Model Management** supports automated operations in the models. These operations include model validation, comparison, generation, merging, and transformation. In the development of critical systems, MDE allows unambiguous expression of requirements and architecture, and provision of automated support for system development and safety assessment (JOHNSON et al., 1998).

Automation is the most effective technological means for improving productivity and reliability, e.g., complete code generation, in which modeling languages take the role of implementation languages. The existing model-based techniques and tools have achieved levels of maturity that enable the practical usage of MDE in large-scale industrial applications. Modern code generators and related technologies can produce code whose efficiency is comparable to or even better than hand-crafted code (SELIC, 2003). The

models are classified into structured or formal, which have a well-defined meta-model, or non-structured, which have not been defined based on a meta-model. The Meta-Object Facility (MOF) and Eclipse Modeling Framework (EMF) (ECLIPSE, 2018a) are development infrastructures for domain-specific language and modeling tools.

Figure 2.5 illustrates the OMG Meta-Object Facility (MOF) infrastructure, comprising four hierarchical levels. The **M0** represents concrete entities, i.e., the instantiation of a meta-model, i.e., a model. The **M1** represents the concepts associated with a domain-specific meta-model built upon abstractions defined in **M2** and **M3**. **M2** represents abstractions defined in UML and **M3** is the highest abstraction level used to define meta-models, e.g., MOF entities, their attributes, and relationships (ATKINSON; KUHNE, 2003).

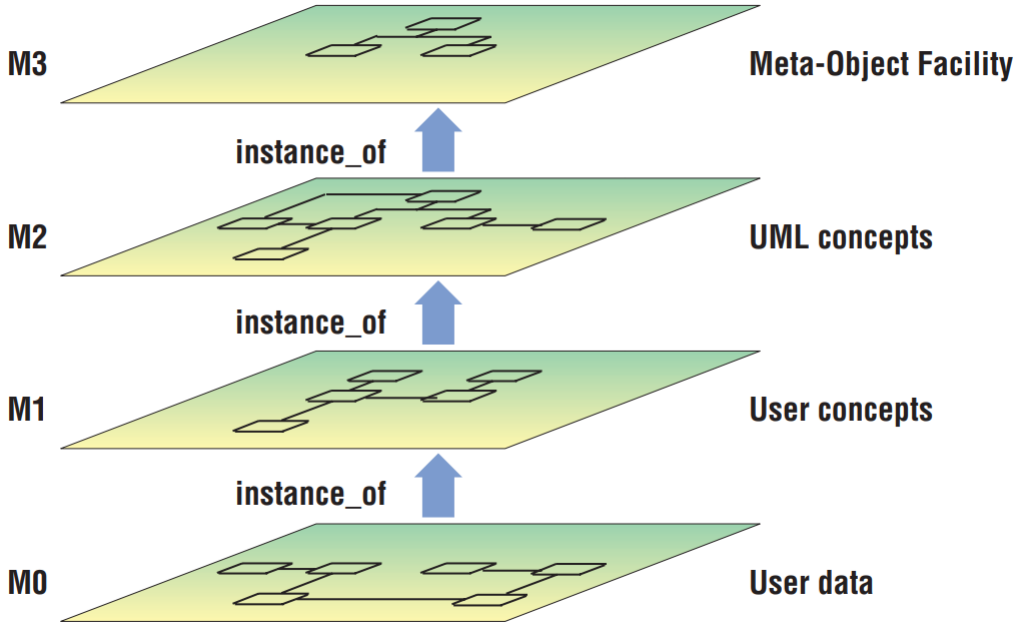

Figure 2.5 – OMG Meta-Object Facility (ATKINSON; KUHNE, 2003)

2.6 MODEL-BASED SAFETY ANALYSIS

Model-Based Safety Analysis (MBSA) is an approach where the system and safety engineers share a common system model created and incremented using a model-based development process. This model used for both system and safety engineering can reduce the cost, improve the quality of the safety analysis, and provide support for automating this process. Within the safety life-cycle safety engineers must perform analysis (e.g. Fault Tree Analysis) based on information originated from several sources, including informal design models and requirements documents. However, these analyses are highly subjective and dependent on the skill of the engineer. For instance, Fault trees which are one of the most common techniques used by safety engineers can differ in substantive ways depending

on the engineer's skill. Even in the final product after review and consensus, it is unlike the result will be complete, consistent, and error-free due to informal models used as the basis for the analysis. The lack of precise models of the system design and its failure modes pushes the safety analysts to devote much more time and effort to gather information about the system and embedding this information in the safety analysis artifacts. This problem is addressed by using formal models for the system in development and safety life cycle. In model-based development, various activities such as simulation, verification, testing, and code generation are based on a formal model of the system under development. The MBSA is an extension of the MDE where the safety analysis activities in addition to the traditional development activities are incorporated into the model-based development. MBSA operates on a formal model describing both the nominal system behavior and the fault behavior (JOSHI et al., 2006).

2.7 GSN ASSURANCE CASES

This section presents two graphical notations for representing assurance cases and their features. Section **2.7.1** describes the Goal Structuring Notation (GSN) and Section **2.7.2** describes the GSN Pattern Extension.

2.7.1 Goal Structuring Notation

Goal Structured Notation (GSN, 2018) is a graphical notation to support the specification of Assurance Cases. Assurance cases have been adopted in a growing number of industries in Europe in domains such as defense, aerospace, nuclear, and railway, e.g., *Eurofighter Aircraft Avionics Safety Justification*, *U.K. Dorset Coast Railway Re-signalling Safety Justification* and *Submarine Propulsion Safety Justifications* (KELLY; WEAVER, 2004).

The GSN base elements are Goal, Solution, Strategy, Context, and Undeveloped Goal. Figure 2.6 illustrates the graphical representation of each one of these elements. GSN provides a hierarchical goal structure that decomposes goals into sub-goals. This allows engineers to demonstrate how the goals are successively broken down into sub-goals until reaching a point where they are supported directly by the available evidence (solutions), i.e., whatever artifact supports the arguments/assumptions that the system is acceptably safe, e.g., test cases and FTA.

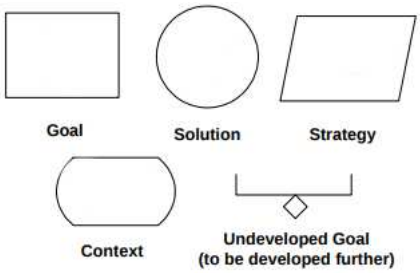

Figure 2.6 – Elements of GSN notation (GSN, 2018).

Figure 2.7 shows an example of goal structure in GSN. It argues that the system is considered safe based on the safety of system functions that have been implemented. The main Goal (G1) arguing that "MySystem is safe" is addressed by arguing that "All system functions are safe" (S1). Context C1 gives the information that Strategy S1 can only be executed in the context of the system functions that have been implemented. In this example, function1 and function2 have been implemented. Thus, the sub-goals G2 and G3 have been constructed arguing that these functions are safe and supporting this affirmation with Solutions, which are test cases (TC1, TC2).

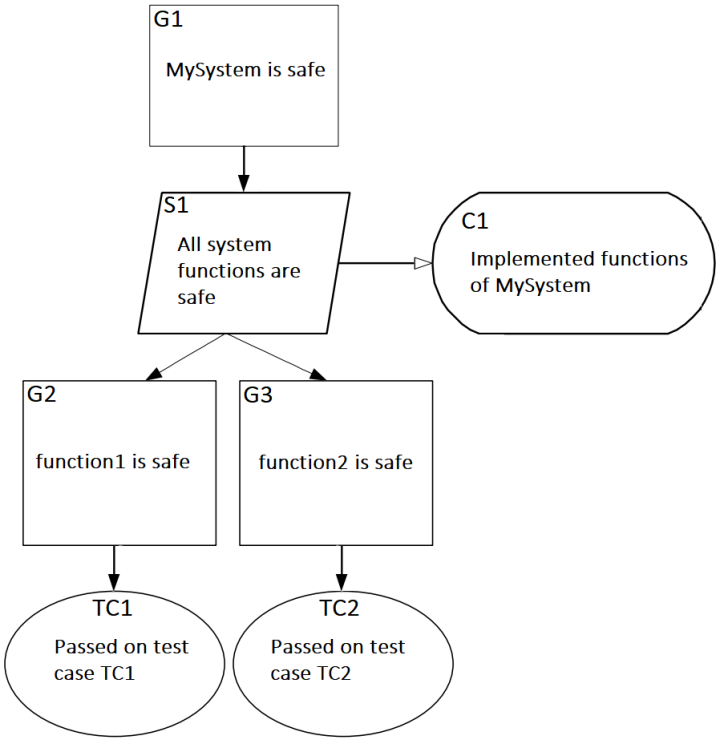

Figure 2.7 – An Example of GSN Goal Structure

2.7.2 GSN Pattern Extension

GSN Pattern Extension provides support for specifying assurance case patterns. In this extension, the abstract *terms* should be represented between brackets, i.e., {*Term*}, as described in the example. The GSN Pattern extension supports two types of abstractions:

structural abstraction, which supports the generalization of relationships such as one-to-one and one-to-many, and **element abstraction** that allows the generalization or postponing details of an element in the argument structure.

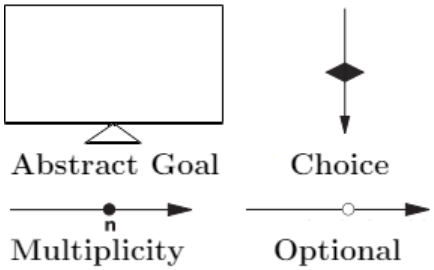

Figure 2.8 – GSN Abstractions (GSN, 2018).

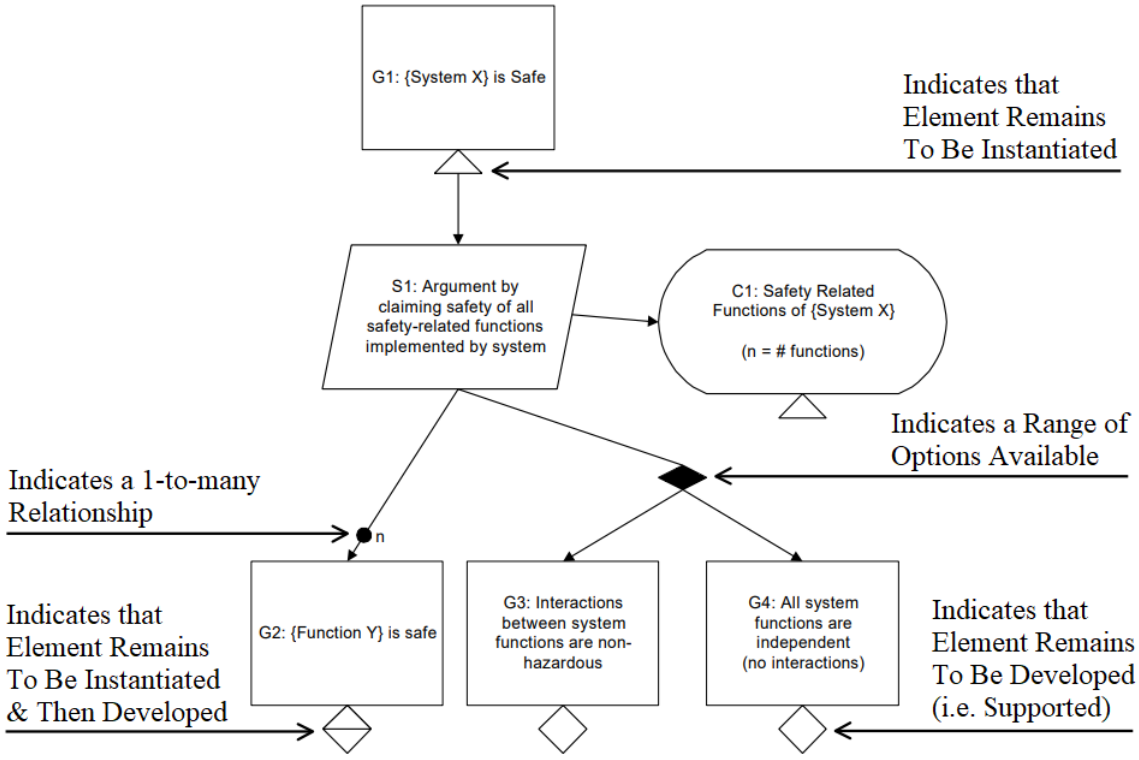

Figure 2.9 – GSN Pattern Representation Example (KELLY; MCDERMID, 1997).

GSN pattern extension defines constraints types with the visual notation to represent structural and element abstractions (Figure 2.8): **multiplicity**, which adds multiplicity to a relationship between goal and sub-goals; **Optional** that means that a relationship between a GSN Goal and sub-goal can be optionally instantiated; and **choice**, which specifies a choice that has to be made when the source element is instantiated, i.e., what target elements will be instantiated after the instantiation of a source element (ACWG, 2022). GSN **multiplicity**, **optional** and **choice** constraints provide information about how to instantiate the elements associated with abstract terms. Figure 2.9 shows an example of a GSN pattern representation adapted from (KELLY; MCDERMID, 1997).

The purpose of this pattern is to show the safety of a system (*G1*) regarding each one of its safety-related functions (*S1*, *C1*). The terms *System X* and *Function Y* are placeholders for specific systems information. A **multiplicity** is associated with the goal *G2* due to the number of functions that a system may have. The **choice** is associated with *G3* and *G4* to indicate the behavior of these functions, i.e. they can be either non-hazardous (*G3*) or independent (*G4*).

2.8 STRUCTURED ASSURANCE CASE METAMODEL (SACM)

SACM is a standard for assurance case modeling languages developed by specifiers of existing system assurance approaches, built upon the collective knowledge and experiences from safety and/or security practitioners over the last two decades (WEI, Ran; KELLY, Tim P; DAI, Xiaotian, et al., 2019). SACM provides the following features: modularity; multiple language support; controlled vocabulary; describing the level of trust in arguments; and counter-arguments; traceability from evidence to the artifact; and automated assurance case instantiation. The SACM meta-model is divided into different sets of elements (OMG, 2021). Each set has more specific purposes, and when these sets are grouped, they compose the SACM meta-model illustrated in Figure 2.10. The colors in the figure are used to group these sets of elements. The groups are structured into assurance case base classes, assurance case packages, terminology classes, argumentation metamodel, and artifact metamodel. Each of these groups will be discussed in the following sections.

2.8.1 Assurance Case Base Classes

Assurance case base classes (Figure 2.11) express the foundation concepts and relationships of base elements of the SACM meta-model and are utilized, through inheritance, by the bulk of the rest of the meta-model, (OMG, 2021). *SACMElement* contains the basic properties shared among all elements of a structured assurance case. Each *SACMElement* has a *+gid*, a unique identifier within the scope of a model instance, an *+isCitation* Boolean flag indicating whether it cites another element, and an *+isAbstract* Boolean flag to indicate whether the *SACMElement* is considered to be abstract. The *+isAbstract* flag is used to indicate whether an element is part of an assurance case pattern or template. *ModelElement* refines *SACMElement* with a *+name* and references to *UtilityElements* such as *Description* and *ImplementationConstraint*.

LangString is the SACM format for description. It has the same purpose of a String with the additional specification of the language (*+lang*) used for its *+content*. A *ModelElement* can contain a *Description* that provides its content, e.g., a *Description* that provides the text of a *Claim*. *ExpressionLangString* extends *LangString* to denote a structured expression, which can be (optionally) used to refer to an *ExpressionElement* (*Term* or *Expression*) in the *TerminologyPackage*. The *+citedElement* is a reference to

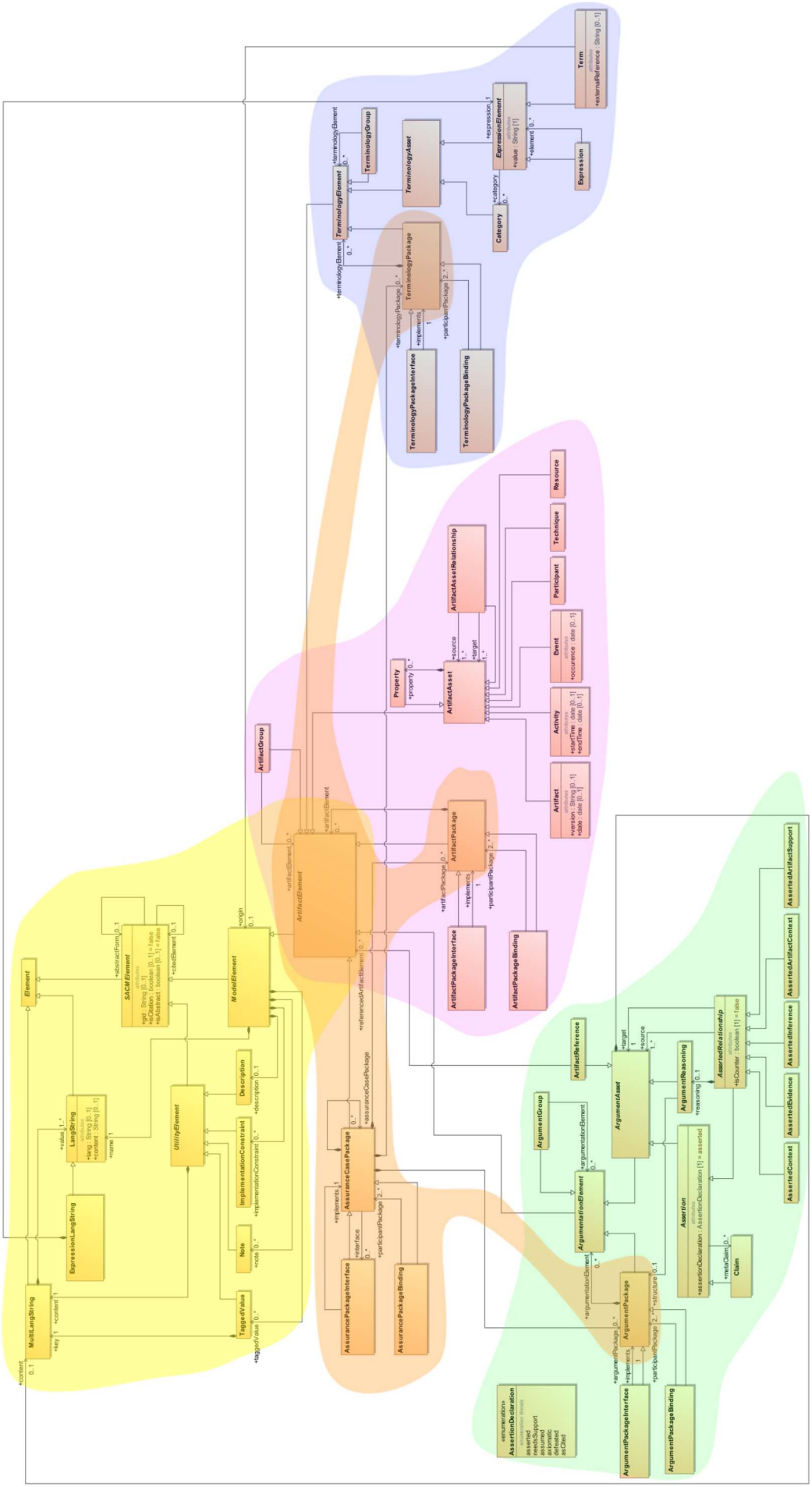

Figure 2.10 – SACM Metamodel (OMG, 2021).

another *SACMElement* that the *SACMElement* cites. If *+citedElement* is not empty, *+isCitation* must be true. Finally, a concrete *SACMElement* can also have an optional *+abstractForm* reference to an abstract *SACMElement* to which this element conforms. Thus, the property *+abstractForm* is used in concrete elements (*+isAbstract = false*) to refer to abstract elements (*+isAbstract = true*)

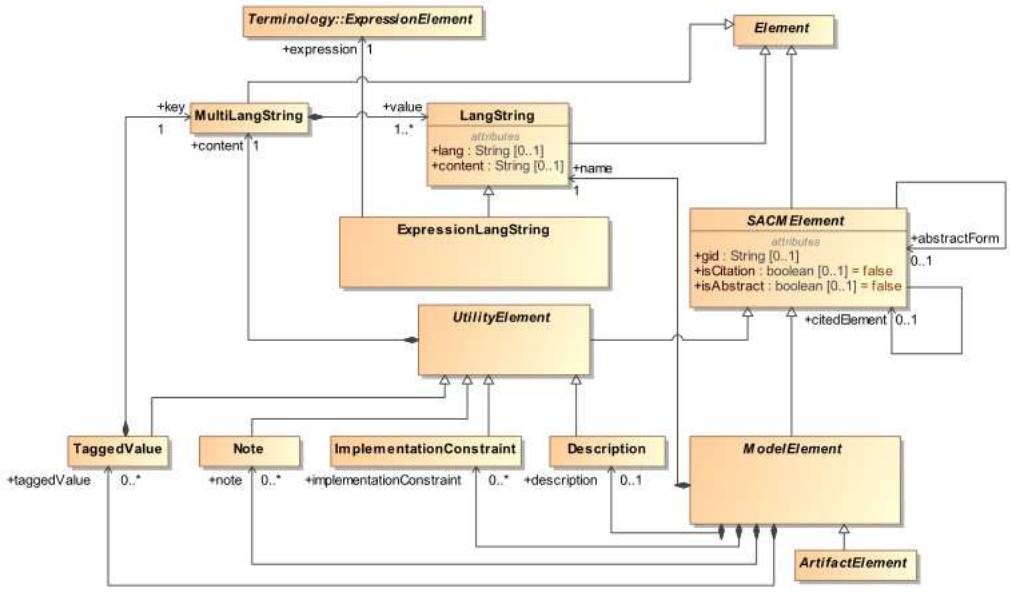

Figure 2.11 – SACM Assurance Case Base Classes.

The *ModelElement* extends *SACMElement* and it is the base element for the majority of SACM modeling elements. A *Description* can be represented in any language via *MultiLangString*. A *MultiLangString* provides a means to describe things using different languages. It contains a list of *LangString* used to specify the languages and the descriptions in the languages. A *ModelElement* can contain zero or more *ImplementationConstaints*. An *ImplementationConstraint* specifies the details of a constraint that must be satisfied to convert a referencing *ModelElement* from *+isAbstract = true* to *+isAbstract = false*. For example, in the context of a SACM assurance case pattern fragment, an element will need to satisfy the implementation rules of the pattern to be instantiated in a concrete assurance case model.

The language used in the specification of an *ImplementationConstraint* is not limited to computer languages (e.g., OCL, EOL, SQL). In the SACM *ImplementationConstraints* can even be described in natural language. However, natural language constraints cannot be used in the automatic instantiation of assurance case patterns due to their non-deterministic nature leading to challenges and difficulty in interpreting them to instantiate the abstract elements. Thus, deterministic computer languages *ImplementationConstraints* allow users to query information available in external artifacts (e.g., ODE design, analysis models), using a query language such as OCL, which can be used to instantiate Terms and other

abstract elements of a SACM pattern. A *ModelElement* can also contain *Notes*, to hold additional generic and unstructured information other than descriptions, and *TaggedValues*. A *TaggedValue* is a simple key/value pair that can be attached to any SACM element. It is an extension mechanism to allow users to add attributes to an element beyond those already specified in SACM. *ArtifactElement* extends *ModelElement* and acts as the base class for elements in other SACM packages. All elements that extend from *ArtifactElement* are considered artifacts and can be referenced via *SACMArgumentation::ArtifactReference*.

2.8.2 Structured Assurance Case Terminology Classes

Structured assurance case terminology classes define the concepts of terms and expressions providing the formalism to create them. *Term* can be abstract if the *+isAbstract* property is set true, or concrete if *+isAbstract* is false. Abstract *Terms* can be considered placeholders for concrete terms, i.e., in the assurance case pattern instantiation the abstracts *Terms* become concrete *Terms*. The *Expression* is used to construct expressions composed by others *ExpressionElements*, i.e., *Terms* or other *Expressions*.

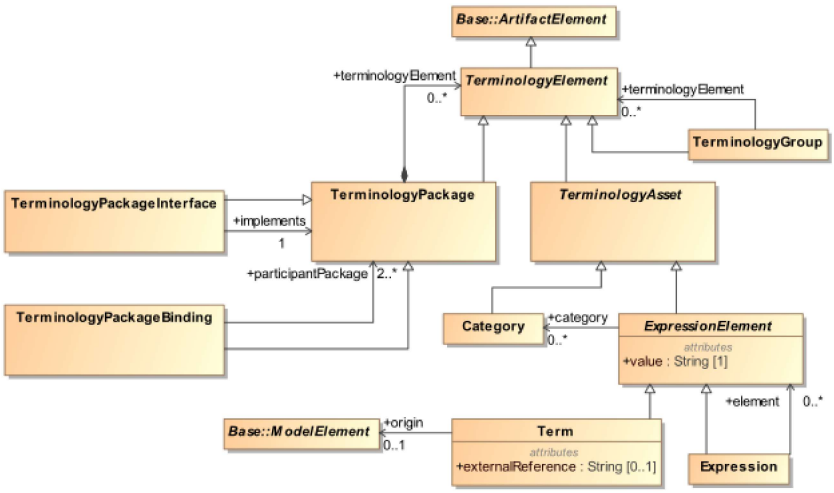

Figure 2.12 – SACM Structured Assurance Case Terminology Classes.

2.8.3 Artifact Metamodel

Artifact metamodel is used to manage corresponding objects that are available, e.g., an artifact which is a test case linked to the requirement that validates the test case once it has already been created. Any elements in the meta-model that extends to *ModelElement* can be considered an *ArtifactElement*, because the *ModelElement* extends the *ArtifactElment*. Thus, the assurance case elements themselves can be used as artifacts.

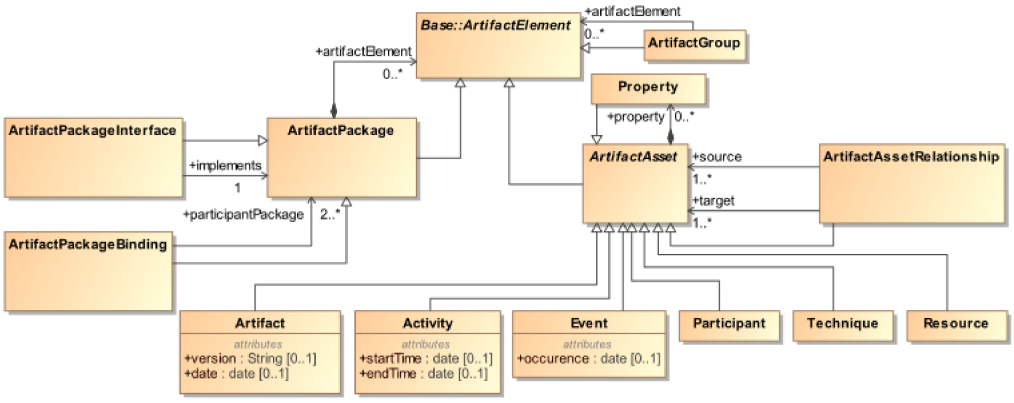

Figure 2.13 – SACM Artifact Metamodel.

2.8.4 Argumentation Metamodel

Argumentation metamodel defines the necessary concepts to model structured arguments, e.g., elements, relationships among them, and their properties (Figure 2.14). Structured arguments are represented in SACM through *Claims*, citations of artifacts or *ArtifactReferences* (e.g., Evidence and Context for Claims), and the relationships between these elements expressed via *AssertedRelationships*. The *AssertedRelationship* is an abstract meta-class for different types of relationships with their respective meaning and usage. *AssertedInference* is used to associate one or more *Claims* to another *Claim*. *AssertedContext* and *AssertedEvidence* are used to connect one or more *ArtifactReferences* to a *Claim*. In addition to these elements, SACM supports the provision of an additional description of the reasoning (*ArgumentReasoning*) associated with inferential and/or evidential relationships.

Figure 2.15 illustrates the visual representation (concrete syntax) of the main SACM Argumentation elements. A *Claim* is represented by a rectangle where the claim statement (description) is written within the rectangle, and a unique identifier is placed at the top left corner of the rectangle. An inferential relationship between *Claims* (*AssertedInference*) is represented using a line with a solid arrowhead and a solid dot in the middle of the line that can be used as a connection point. An *ArgumentReasoning* used to describe non-obvious relationships between *Claims* and *ArtifactReferences*, is represented by an annotation symbol. It can be attached to an *AssertedInference*, which connects *Claims*, or to an *AssertedEvidence*/*AssertedContext* which connects *Claims* to *ArtifactReferences*. *ArtifactReference* is represented using a document symbol, to provide a clue to its meaning (as an artifact), with an arrow placed on the top right of the symbol to denote a reference. *ArtifactReferences* are represented by rectangles nodes with the *+gid* in the top left corner and *Description* in the middle. While *ArgumentReasoning* is represented by an open semi-rectangle node with the *gid* in the top center and *Description* in the middle.

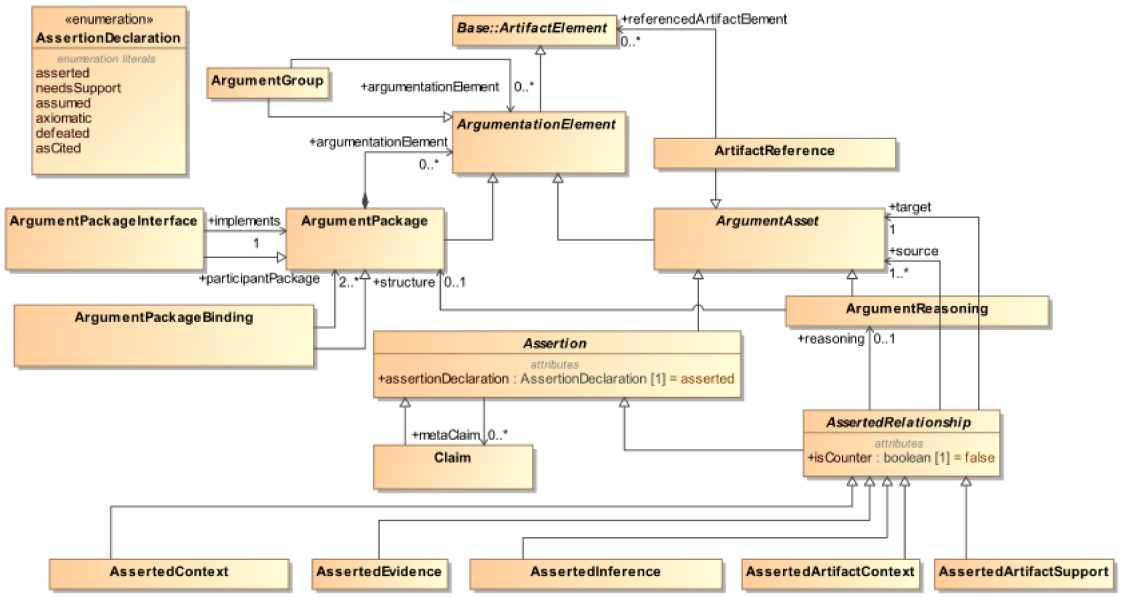

Figure 2.14 – SACM Assurance Case Base Classes.

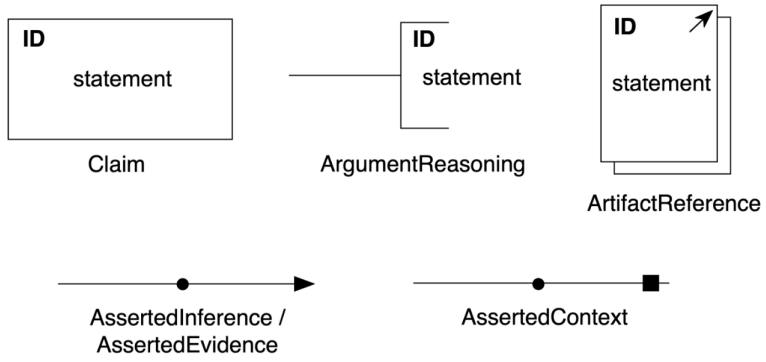

Figure 2.15 – SACM Argumentation package elements (OMG, 2021).

A *Claim* can be supported by one or more *Claims*, and the relationships between them can be defined using an *AssertedInference* relationship. In no obvious relationships, a *ArgumentReasoning* can be used to provide a further description of the reasoning involved. A *Claim* may cite a reference to contextual and/or evidential information, this can be done through *ArtifactReference* elements. However, to differentiate between them it is necessary to identify the type of relationship to which the *ArtifactReference* and *Claim* are related. Contextual information is using *AssertedContext* and *AssertedEvidence* relationships respectively, (SELVIANDRO; HAWKINS; HABLI, 2020).

Figure 2.16 shows an excerpt of Hazard Avoidance (KELLY; MCDERMID, 1997) argument pattern in the SACM visual notation. This pattern provides the abstract structure for arguing a given "{SystemX} is safe" (*C1*) based on the avoidance of each identified hazard (*C2*). *C1* abstract *Claim* is represented by a dashed rectangle. *C2* abstract *Claim* is represented by a dashed rectangle decorated with a three-dot symbol to denote this claim *needs* further supporting evidence or argumentation. *C1* and *C2* abstract

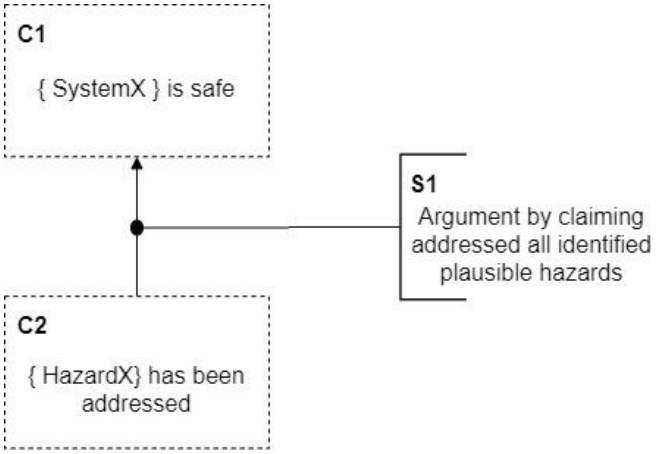

Figure 2.16 – Excerpt of the Hazard Avoidance pattern (KELLY; MCDERMID, 1997).

Claims are connected via an *AssertedInference* relationship with an *S1* element describing the reasoning for decomposing *C1* into *C2* sub-claim arguing each identified "{HazardX} has been addressed". Abstract *Terms* are within brackets in the *Expressions* that describe each abstract *Claim*. Each abstract *Term* is a placeholder referencing information from an external artifact (design, analysis, or process model) required to instantiate it in a concrete assurance case model.

SACM supports the specification of *ImplementationConstraints* to abstract elements of an assurance case pattern. However, it is not possible to specify executable SACM assurance case patterns (i.e., assurance case patterns that can be automatically instantiated based on analysis, design, and process models) with explicit links between evidence referenced within abstract Terms (placeholders) and external artifacts. These artifacts can be ODE Design, Hazard Analysis and Risk Assessment (HARA), Failure Logic, Fault Tree Analysis, and Security Analysis models, which constitute the Digital Dependability Identity (WEI; KELLY, et al., 2018) of a CPS or CPS component. The explicit links, i.e., traceability, between abstract Terms (placeholders) and artifacts are necessary to enable the automated instantiation and reasoning of safety/security assurance claims of open and adaptive CPSs at runtime. In addition to the lack of traceability, there is still a need to add semantics to the concept of *ImplementationConstraints* to support management and the systematic reuse of SACM assurance cases in the same way patterns extensions (KELLY; MCDERMID, 1997) in GSN assurance cases. Therefore, it is needed to provide semantic support for representing the concepts of *n-any*, *optional*, *alternative* structures with *traceability* in the SACM. These features provide the basis for the automated synthesis of executable assurance case patterns. By implementing these features, it is possible to specify: *i)* structural relationships between abstract SACM argumentation elements in an assurance case template; *ii)* generalization/specialization relationships between abstract argumentation elements; *ii)* traceability links between abstract terminology elements and information from EDDI artifacts. This can be considered a preliminary solution for

the realization of the concept of Executable Digital Dependability Identity and runtime assurance (WEI; KELLY, et al., 2018; DEIS, 2020).

Hence, it is needed to support the specification of complex dependence relationships between abstract *Terms* of a SACM assurance case template, e.g., the instantiation of HazardX abstract *Term*, from Hazard Avoidance pattern illustrated in Figure 2.16, is dependent upon the *System.name* information retrieved from the design model required to convert SystemX *Term* from *isAbstract = true* to *isAbstract = false*. The potential of computer languages such as OCL and EOL in providing semantic support for *ImplementationConstraints* to connect abstract terminology and argumentation elements from SACM assurance case patterns to external artifacts was not fully exploited yet to enable the automated reasoning of safety/security assurance of CPSs at runtime.

2.9 RUNTIME ASSURANCE

Since most open and adaptive CPS domains are safety-critical, it is imperative to ensure the safety, security, and other dependability properties of a CPS/CPS component face unknowns and uncertainties, introduced by Artificial Intelligence, at runtime (WEI, R.; KELLY, T. P.; DAI, X., et al., 2019; WEI; KELLY, et al., 2018). Thus, assurance cases are expected to be exchanged, integrated, and verified at runtime to ensure the dependability of CPSs. However, established assurance approaches, e.g., Goal Structuring Notation (GSN) (GSN, 2018), Claims-Arguments-Evidence (CAE) (BLOOMFIELD, R.; BISHOP, P., 2010), and standards designed to address standalone systems, building a complete understanding of the system and its environment at design time, are insufficient to assure the dependability of CPSs. CPSs are loosely connected and come together as temporary configurations of smaller systems that may dissolve and give place to other configurations. The key problem in assessing the safety and security of CPS is that it is almost impossible to anticipate the concrete CPS structure (configuration), capabilities, and environment at design time. The configurations a CPS may assume over its lifetime are unknown and potentially infinite. Moreover, state-of-the-art dependability analysis techniques require prior knowledge of the CPS configurations at the design phase, which provides the basis for the analysis of systems. Thus, existing assurance approaches can limit runtime flexibility, and cannot cope with the complexity of CPSs (WEI, R.; KELLY, T. P.; HAWKINS, R., et al., 2017).

Assurance cases at runtime are needed to enable open and adaptive systems to inspect the assurance case of other systems at runtime to evaluate if they are safe and/or secure to interact with (TRAPP; SCHNEIDER; LIGGESMEYER, 2013). This is needed for autonomous systems in which some claims about safety/security can only be instantiated at runtime due to the uncertainties introduced by some of its components (e.g., AI) (WEI; KELLY, et al., 2018). A runtime assurance case allows autonomous CPS

to infer their safety and security. When systems adapt their behaviors in response to changes in their environment, the evidence (e.g., verification and validation models - V&V) may become invalid. Hence, runtime V&V models may support reinstating the evidence used (referenced) by assurance case models (WEI, R.; KELLY, T. P.; DAI, X., et al., 2019).

In a first step towards runtime system assurance, SACM has been used in the DEIS project (DEIS, 2020) as a backbone for its Open Dependability Exchange meta-model (ODE) within the Digital Dependability Identity (DDI). ODEs system models are used to assure the dependability of Cyber-Physical Systems at runtime (Figure 2.17). The runtime assurance is achieved through an assurance case with traceability links to the design and analysis ODE models (i.e. evidence) within the argumentation structure. At runtime, these models may change based on data from sensors. Then, the certification algorithm periodically evaluates the assurance case. Thus, if any ODE model changes at runtime the re-certification of the assurance case is executed. SACM provides the foundations for argumentation about the safety and/or security of a wide range of systems. It also supports referencing evidence of their arguments at both design time and runtime,(WEI; KELLY, et al., 2018).

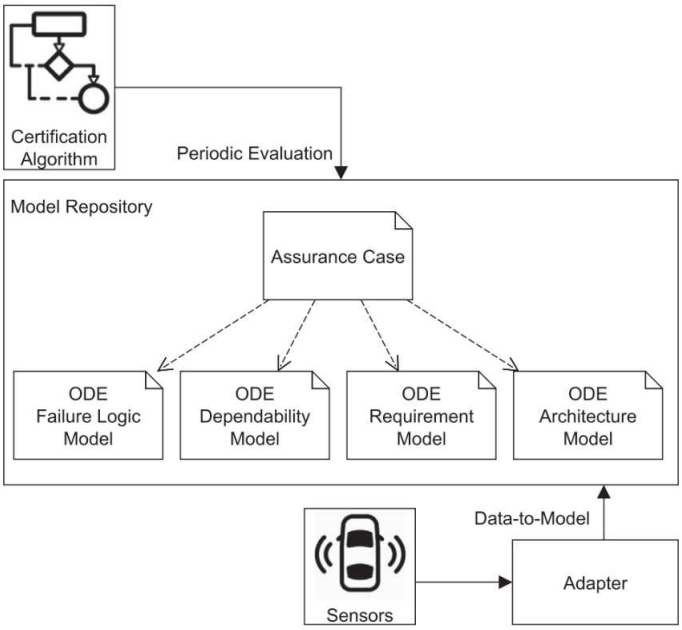

Figure 2.17 – Runtime assurance overview (WEI, R.; KELLY, T. P.; DAI, X., et al., 2019).

The assurance of CPS and/or CPS component dependability properties demands a paradigm shift from parts of the system assurance process activities at design time to runtime where uncertainties can be resolved dynamically (WEI; KELLY, et al., 2018). With this shift, there will be a transition from the current design time assurance cases produced from manually created artifacts to assurance case models that can be automatically synthesized and evaluated at runtime. To achieve this goal, it is necessary to equip a CPS or a CPS component with all the information that uniquely describes its dependability

characteristics (design, analysis, and process models) within a Digital Dependability Identity (DDI) (WEI, R.; KELLY, T. P.; HAWKINS, R., et al., 2017). Thus, DDIs produced at design time provide the basis for automated integration of components into systems at development time and dynamic integration of independent systems into systems of systems at runtime.

2.10 DIGITAL DEPENDABILITY IDENTITY

A Digital Dependability Identity (DDI) is an evolution of classical modular dependability assurance models. DDI formally integrates several separately defined dependability aspect models allowing for comprehensive dependability reasoning (DEIS, 2020). DDIs are produced during design, and certified when the component or system is released. Then, it is continuously maintained over the lifetime of a component or system. Therefore, it requires continuous traceability between a SACM safety argument and safety-related evidence models stemming from hazard and risk analysis, functional architecture, safety analysis, and safety design concepts (Figure 2.18). Although the SACM argument is about the safety property of a system, the DDI concept is general enough to equally apply it to other dependability properties such as security, reliability, or availability, (REICH; ZELLER; SCHNEIDER, 2019).

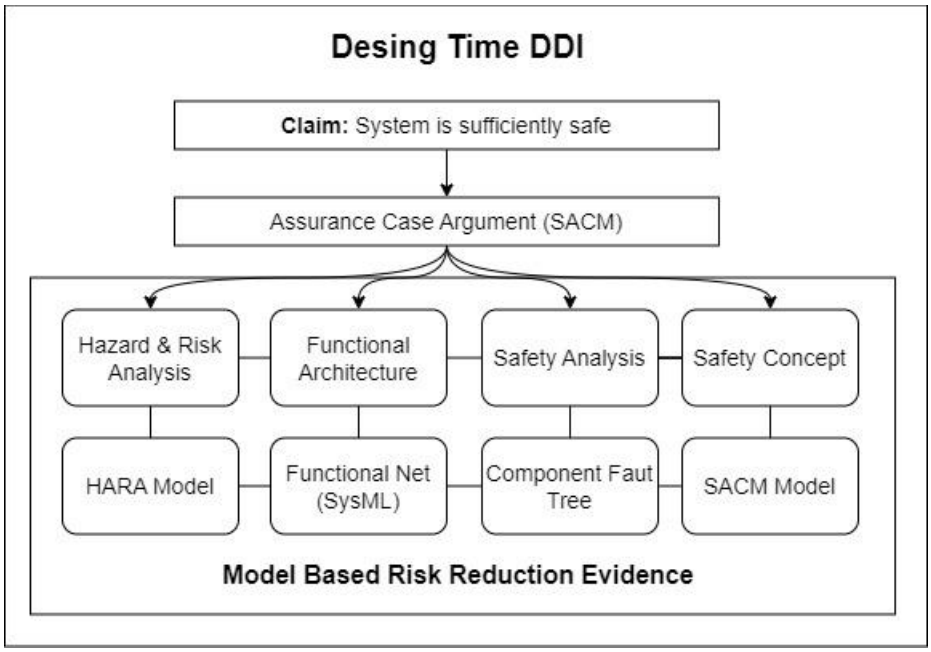

Figure 2.18 – Digital Dependability Identity

A DDI comprises all the information that uniquely describes the dependability characteristics (design, analysis, and process models) of a CPS or a CPS component (WEI, Ran; KELLY, Tim P; HAWKINS, Richard, et al., 2018). DDIs provide a formal basis for automated and dynamic integration of independent systems into systems of systems in the field at runtime. The Open Dependability Exchange (ODE) is a metamodel for

representing DDIs. Thus, the ODE provides the basis for representing and exchanging development and safety information (e.g., FME(D)A, FTA, and Markov Chains) between open-adaptive CPSs and CPS components. The SACM metamodel is the backbone of the ODE, providing the formal traceability between assurance cases and DDI information. Therefore, SACM assurance case patterns can be instantiated based on the information provided by CPS and CPS component DDIs.

2.11 OPEN-DEPENDABILITY EXCHANGE METAMODEL (ODE)

The Open-Dependability Exchange metamodel (ODE) comprises several packages as illustrated in Figure 2.19. The ODE and its core features are at the highest level of abstraction and aggregate the Dependability, Design, and Failure Logic packages. The Dependability package elements represent artifacts created/required during the early phases of the development process. For instance, it can be used to specify the system development standards and identify functional and non-functional requirements and their prioritization through early safety activities such as Hazard Analysis & Risk Assessment (HARA). The design process is supported by the Design package where it is possible to define the primary entities of the system, and how they relate to each other concerning composition and their interactions. The Failure Logic package provides the basic concept of safety analysis techniques to evaluate whether the design sufficiently avoids, removes, or mitigates the identified hazards. To investigate the relationship between causes and safety measures the cause analysis is progressively broken down to lower-level elements of the architecture.

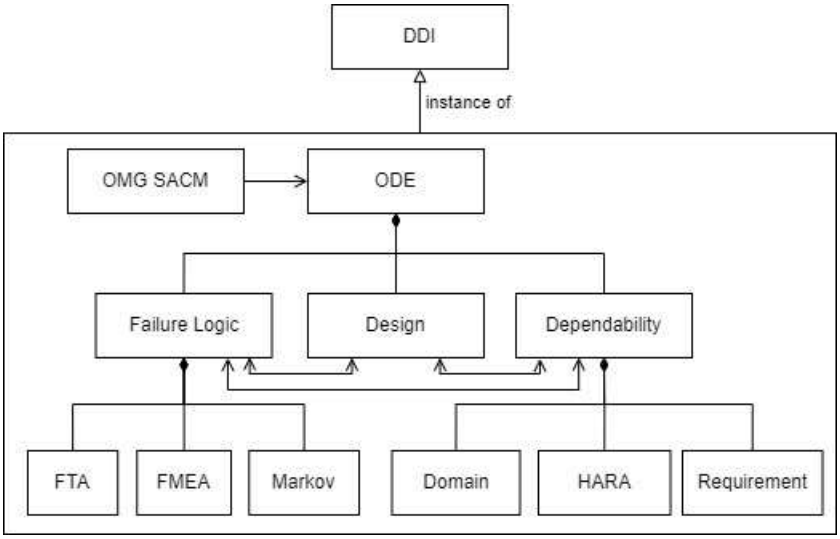

Figure 2.19 – Overview of the Open Dependability Exchange Metamodel

The ODE supports three different safety analysis techniques that are relevant for the industry. These techniques are the Fault Tree Analysis (FTA), Failure Mode and Effects Analysis (FMEA), and Markov analysis which are within the Failure Logic

package. The information represented in the ODE can be used as supporting evidence in an assurance case. Therefore, the SACM is used to represent the argumentation structure of the assurance case linking all relevant artifacts (e.g., hazards, fault trees, and failure logic models) represented in the ODE to elements of the argumentation. Thus, the SACM is the backbone of the ODE and the DDI concept, structuring and relating all system models and information, (ZELLER et al., 2023).

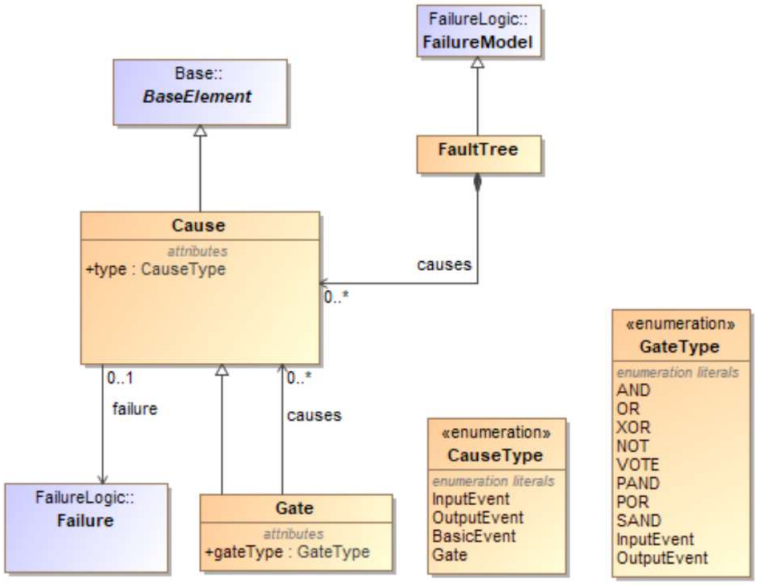

Figure 2.20 – ODE::FailureLogic::FTA Package (DEIS, 2020).

The ODE **FTA Package** meta-model (Figure 2.20) captures the information from Fault Tree Analysis techniques. The **FTA Package** element extends the FailureLogicPackage element enabling references to relevant elements (e.g. InternalFailures). The Gate element captures the logical relationship that associates the various types of events represented within a fault tree structure. Such events correspond to input, output, and internal failures captured in the FailureLogic package. Gates represent the top and intermediate events of fault trees enabling the representation of their hierarchical structure through the *+causes* property. The basic events of fault trees are represented by Cause elements with its *+type* set to InputEvent, OutputEvent, or BasicEvent, (DEIS, 2020).

2.12 RELATED WORK

This section presents the works related to this study. Section **2.12.1** explores a weaving model-based approach for the automatic instantiation of assurance case patterns. Section **2.12.2** describes a table-based assurance case pattern instantiation process. Section **2.12.3** presents a modeling tool for the specification of SACM-compliant assurance cases and patterns.

2.12.1 Weaving-Model Based Instantiation

A weaving model is the key to this approach. This model links the referenced information within meta-models to the GSN argument patterns. The primary objective of the weaving model is to handle fine-grained relationships between elements of distinct models, establishing links between them. The mapping between models themselves are also formal models increasing the expressiveness and flexibility. However, aside from the pattern and the desired system or dependability model, this third model called the weaving model must be specified either manually or automatically (i.e. transformation). In this approach, the model captures the dependencies between roles in GSN patterns and individual/multiple reference information metamodels. Thus, information from external models can be used for the automatic instantiation of the GSN argument, (HAWKINS; HABLI, et al., 2015).

Figure 2.21 shows the overview of the instantiation process using the weaving model approach. First, the GSN Pattern is specified. Then GSN roles are mapped to the Reference Information Metamodel generating the Weaving Model. The inputs for the instantiation process are the GSN Pattern, the Waving model, and the Reference Information Model, i.e., external artifact. After this process, a GSN Argument is generated comprising information from the external artifacts based on the Weaving Model mapping.

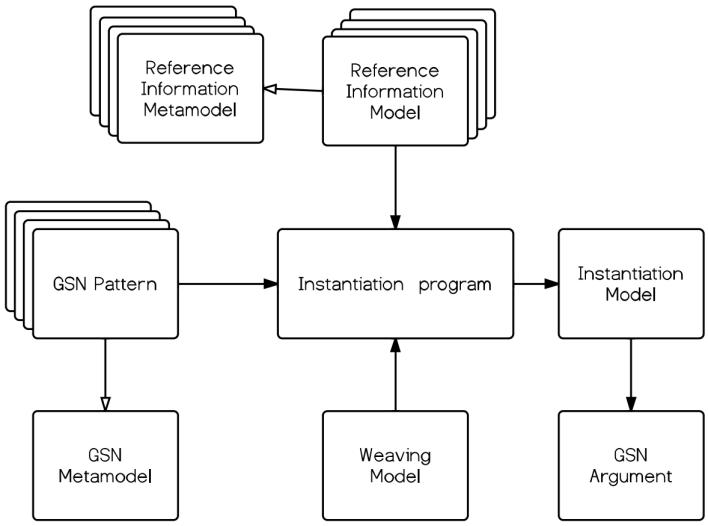

Figure 2.21 – Weaving-Model Based Instantiation (HAWKINS; HABLI, et al., 2015).

2.12.2 Table Based Instantiation

The core of this approach is a generic data table model called P-table and an instantiation algorithm to generate GSN arguments from GSN patterns. The P-table inherits the structure of the intended GSN pattern. The P-table is represented in tabular form where the columns are the data nodes (i.e. Goals, Strategies, Context, Assumptions, Justifications, and Solutions) D, and rows by D $\times$ V pairs, i.e., values for abstract roles

or joins. Thus, the P-table must be specified manually or automatically for each pattern with the respective columns with root values and joins to enable automatic instantiation. A row, therefore, consists of the data that instantiates an upward-closed fragment of the pattern, following the paths of the fragment up until the join. To instantiate a GSN pattern from its P-table each row is processed to create a row instance fragment. This is effectively the creation of zero or more instances of a given data node of the pattern assigning the parameter values within table joins. For each value it is not added just the instantiation of the appropriate data node, also any boilerplate between that node and the preceding data node. Multiple values in the P-table lead to multiple instances of a data node. However, those boilerplate nodes which appear after a multiplicity are only repeated. Instances indices are used to connect nodes to the correct parent when there are such multiples. At any point in the algorithm we identify the "current node" as current, and the pattern root as root, (DENNEY, E.; PAI, G., 2013).

In this table-based approach, the automatic generation of GSN assurance cases starts with the specification of an assurance case pattern followed by the specification of the P-table which are the inputs for the instantiation process. In the instantiation phase, the values within P-table cells are used to instantiate the GSN data nodes (e.g. Goals). Figure 2.22 shows an excerpt of the Requirements Breakdown Pattern described in (DENNEY, E.; PAI, G., 2013).

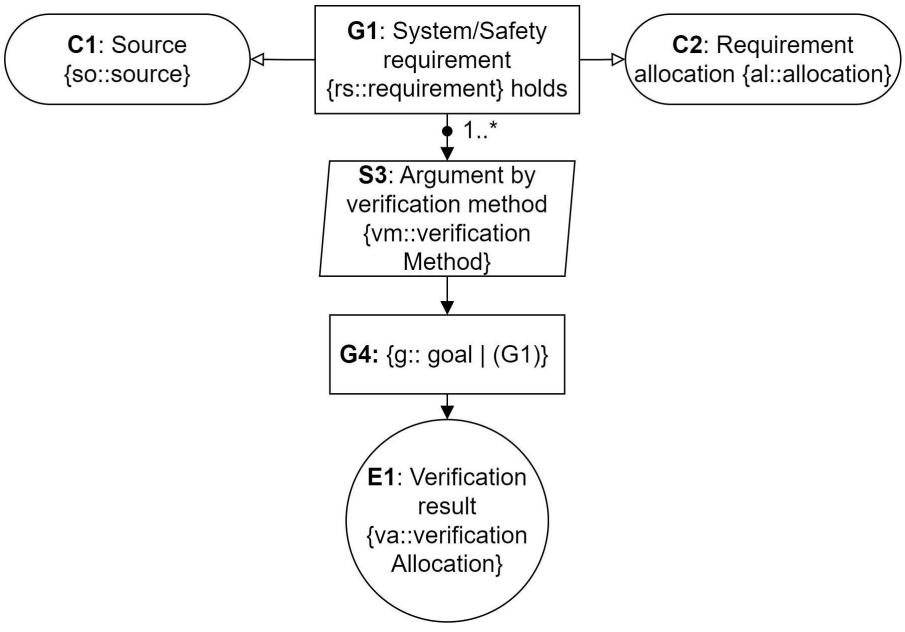

Figure 2.22 – Excerpt Requirements Breakdown Pattern.

Figure 2.23 shows an example of a data table (P-table) where each column is a node of the GSN Requirements Breakdown Pattern and the values and joins are displayed within its rows. The required information to instantiate the assurance case pattern is present in each cell of the table. The structural relation between the instances is given by

the join column.

Parameter Type	*Requirement*	*Lower-level requirement*	*Allocated Requirement*	*Source*	*Requirement Allocation*	*Verification Method*	*Verification Allocation*
Data node / Join	**G1**	**G2**	**G3**	**C1**	**C2**	**S3**	**E1**
	R1	R1.1, R1.2	AR1	S	A	VM11, VM12	VA11, VA12
(S3, VM12)							VA22
(G2, R1.1)						VM1.11, VM1.12	VA1.11, VA1.12
(G2, R1.2)		R1.2.1, R1.2.2	AR1.2				
(G2, R1.2.1)						VM1.2.1	VA1.2.1
(G3, AR1.2)			AR1.21			VM1.2	VA1.2

Figure 2.23 – Example of a populated P-table (DENNEY, Ewen; PAI, Ganesh, 2013).

Finally, Figure 2.24 shows an excerpt of the instance of the Requirements Breakdown Pattern after its instantiation using the P-table data. This example contains the instantiation result for the first two rows of the table. The goal *G1* has been instantiated for the requirement *R1* in context of the source *S* (*C1*) and requirement allocation *A* (*C2*). The Strategy *S3* has been instantiated for each verification method, i.e. *VM11* and *VM12*. The pattern Goal *G4* has been instantiated into Goals *G5* and *G6* for the first verification method *VM11* due to the multiplicity of the verification allocation Evidence (*VA11* and *VA12*). Finally, the pattern Goal *G4* is instantiated into the Goal *G7* for the verification allocation Evidence (*VA22*) regarding the verification method *VM12*.

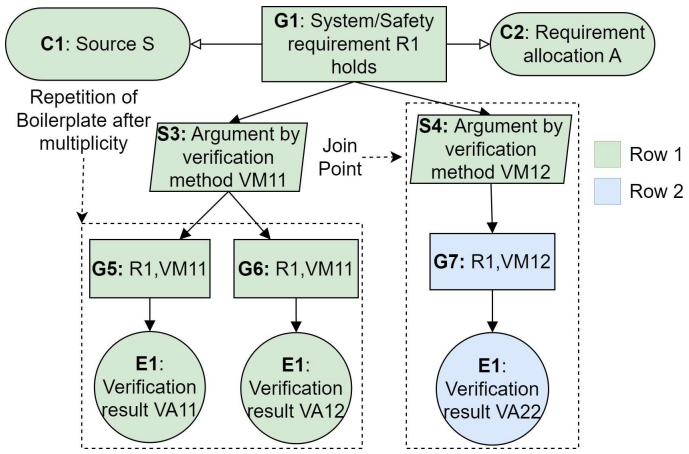

Figure 2.24 – Excerpt Requirements Breakdown Pattern Instance.

2.12.3 Assurance Case Editor

In order to exploit the benefits provided by SACM whilst providing support for existing assurance case approaches (e.g. GSN) an assurance case editor has been developed by (WEI, R.; KELLY, T. P.; DAI, X., et al., 2019). This tool called ACME (Figure 2.25) was implemented using the Graphical Modelling Framework (GMF) (ECLIPSE, 2018b) which supports the creation of editors based on metamodels defined using the Ecore metamodel provided by the Eclipse Modelling Framework (EMF) (ECLIPSE, 2018a). With ACME users can specify an assurance case and assurance case patterns using SACM

or GSN for the arguments. It is the first step towards an integrated modeling environment for SACM.

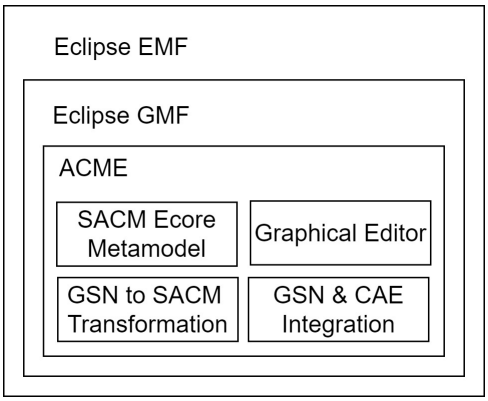

Figure 2.25 – ACME Architecture Overview.

ACME enables existing assurance case approaches to be used in conjunction with SACM exploiting SACM features such as evidence-artifact traceability, controlled vocabularies, and multiple language support. It also introduces the concept of mapping queries within the SACM implementation constraint elements. This feature aims to instantiate placeholders of assurance case patterns based on external artifact information. However, it is still not possible to instantiate assurance case patterns using the ACME tool set. ACME acts as a transitional solution from GSN argument specification to SACM model-based system assurance providing easy-to-use SACM facilities as well as automated model-to-model transformation from GSN to SACM, (WEI, R.; KELLY, T. P.; DAI, X., et al., 2019).

3 SACM PATTERN EXTENSIONS

This chapter describes the pattern extensions proposed in this thesis for assurance case pattern specification in SACM visual notation. Section 3.1 presents the representation of the implementation constraints sub-types. Section 3.2 presents the **Mapping** constraint, Section 3.3 describes the **Multiplicity** constraint, Sections 3.4 and 3.5 show the **Optional** and **Choice** constraints respectively, Section 3.6 presents the **Children** constraint. Finally, Section 3.7 describes the usage and applications of the constraints sub-types in SACM assurance case patterns.

3.1 CONSTRAINT TYPES IN SACM

In the GSN assurance case notation pattern extensions have been proposed defining types of constraints and some general abstractions to provide support for the specification of GSN assurance case patterns, (GSN, 2018). These well-defined constraints allowed the development of automatic instantiation algorithms for GSN assurance case patterns (HAWKINS; HABLI, et al., 2015; DENNEY, E.; PAI, G., 2013). However, in SACM this constraint concept is still too generic, the *ImplementationConstraints* element used for defining the rules for instantiating an element has no defined type or semantics. This lack of definition in the SACM metamodel is a barrier to the automation of the assurance case pattern instantiation. The instantiation algorithm can not interpret natural language constraints or even computer language constraints without considering their meaning. Therefore, in order to fulfill this gap this thesis proposes a pattern extensions for the SACM metamodel to address types of *ImplementationConstraints* as in GSN notation.

SACM pattern extensions add semantics to *ImplementationConstraints* to support the specification of executable assurance case patterns, i.e., assurance case patterns that can be automatically instantiated. The pattern extension is comprised of **Multiplicity(m)**, **Optional(o)**, **Choice(c)**, **Mapping(p)**, and **Children(s)** constraints subtypes. The *m*, *o*, and *c* constraints have the same semantics of GSN Multiplicity and Optionality patterns extensions (HABLI; KELLY, 2010). Mapping constraints are used to relate abstract terms to elements from external artifacts, e.g., design, analysis, or process models, via model queries to obtain model elements (WEI, R.; KELLY, T. P.; DAI, X., et al., 2019). Finally, the Children constraint is used to specify hierarchical relationships between instances of abstract terms in a SACM argument pattern. In this proposed approach a LangString element with its *+lang* slot set with the *constraint type*, and its *+content* slot possibly set with a *query* should be added to the *+content* slot of an *ImplementationConstraint* element to differentiate between *ImplementationConstraints* sub-types (Figure 3.1).

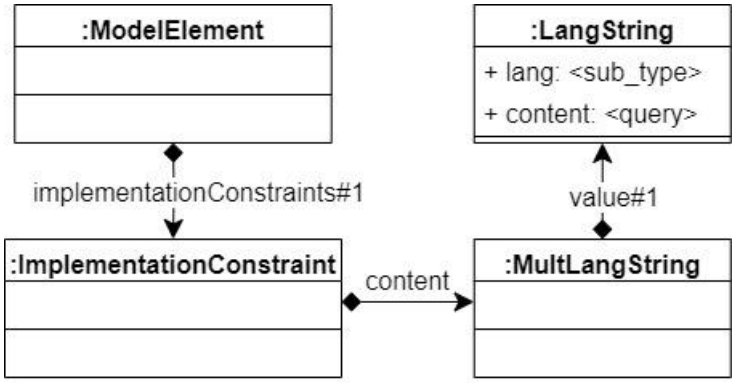

Figure 3.1 – Constraints Subtypes Representation.

Users can associate the implementation constraint sub-types to any SACM ModelElement with no restriction in the used language. The language used in implementation constraints is not limited to computer languages. Natural languages can also be used to describe instantiation rules of implementation constraints, except the instantiation procedure is limited to manual. The automation of assurance pattern instantiation needs tool support and a model management engine to execute the implementation constraints.

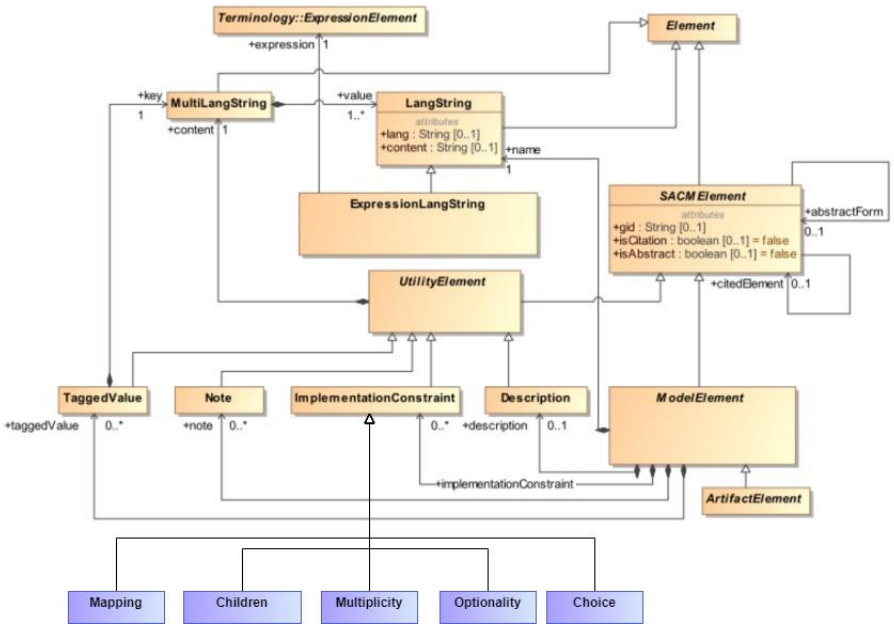

Figure 3.2 – Extended Meta-model Representation.

Computer languages such as Object Constraint Language (OCL) (O.M.G., n.d.) and Epsilon Object Language (EOL) (KOLOVOS; PAIGE; POLACK, 2006) can be used to specify queries for *Mapping* and *Children* constraints when the external artifact is a MOF compliant model. The *Mapping* and *Children* constraints provide traceability by linking abstract terms from SACM argument patterns to external artifacts (design or analysis models). Depending on the external artifact a proper query language should be chosen, e.g., if the artifact is the relational database the SQL language should be used.

Thus, the specification of implementation constraints to elements of SACM argument patterns plays the role of a *weaving model* (HAWKINS; HABLI, et al., 2015) linking abstract terms to elements from external artifacts. The objective of the *weaving model* is to map *abstract Terms* of an assurance case pattern to information within other artifacts. The queries within *abstract Terms* constraints map their values directly to the external artifacts without needing a third model, i.e., *weaving model*. Figure 3.2 shows the SACM Base Classes meta-model extended with the implementation constraints sub-types proposed in this work.

3.2 MAPPING

Mapping (p): can be attached to an abstract *Term t* to impose restrictions on *+value* slots of its instances. **Semantics**: a mapping (*p*) constraint is used to link the *+value* slot of an abstract *Term t* to information from an external artefact. The reference to an external artifact is stored in the *+externalReference* slot of an abstract *Term t*. The constraint *p* denotes that *t* will be instantiated with one or a set of concatenated values retrieved from an external artifact, specified in the *+externalReference* slot from *t*, by executing a computer language model query, e.g., EOL. The return of the query execution must be a list comprising zero or more strings.

Origin relationships between abstract terms can be specified in the argument pattern. This allows the instantiated values of abstract terms to be used as parameters for instantiating other terms. These origin relationships are achieved through the *+origin* slot of abstract terms (Figure 2.12). Therefore, if an abstract *Term t'* originates from another *Term t*, *t'* must have its *+origin* slot set to *t*. Once the abstract terms are instantiated based on the origin relationships, their instances also inherit their origin during the instantiation. This feature provides support for parameterized mapping queries within *t'*, i.e., the values of each instance of *t* can be used to instantiate *t'*.

Example: consider an abstract *Term portY* that originates from a *Term componentX*, and a system design model with components and ports where each port is related to a component. The EOL mapping query *Component.all.collect(c | c.name)* can be added to the abstract *Term componentX* to get the names of all components and instantiate *componentX*. The parametrized EOL mapping query *Component.all.selectOne(c| c.name='$origin').ports.collect(p| p.name)* can be added to the abstract *Term portY* to get all the ports names related to each $origin *componentX* and instantiate *portY*. It is possible because at instantiation time the $origin parameter is replaced by each *+value* of the instances of *componentX* and the resulting queries are executed to generate the instances of *portY*.

3.3 MULTIPLICITY

Multiplicity (m): can be attached to SACM *ModelElements*, e.g., *argumentation* and *terminology expression* elements. **Semantics**: *m* denotes *zero-or-more n-ary* cardinality of an abstract *ModelElement*, e.g., *Claim*, *AssertedRelationship*, *Term*, in an argument pattern. In the same way as GSN (GSN, 2018), *multiplicity* indicates zero or more instances of an abstract *ModelElement* (*Claim*, *Term*) relate to *element property values* retrieved from an *external artifact*. A *m* constraint attached to a *ModelElement* denotes that zero or multiple instances of such an element should be created during argument pattern instantiation. This type of constraint subtype can be attached to various types of abstract elements as described below.

- *Terms*: it indicates that the number of instances of abstract terms with *m* constraint is related to the number of values resulting from *mapping (p)* constraint execution.
- *Expressions*: it indicates that the number of instances of abstract expressions with *m* constraint is related to the number of instances of abstract *Terms* within the expressions *+element* slot.
- *AssertedRelationship*: the multiplicity constraint within these elements has the meaning of determining the relationship instances, their target, and their sources. The relationship instances are determined by instances of the *+target* element. The sources of these relationship instances are all the instances of sources from the abstract relationship that have in their *Description* slot direct reference and/or have a term that originates from one or more terms of the *+target* relationship instance.
- *AssuranceCase*, *Artifact*, and *Argument* packages including binding and interfaces: the multiplicity constraint within these elements has the meaning of determining the packages instances and their contents. The package instances are determined by instances of abstracts *Terms* referenced through *+gid* in the *+content* slot of the *m* constraint. The contents of these package instances are all the instances of elements from the abstract package that have direct or indirect reference to the referenced term instance or have a term that originates from it in their *Description* slot.
- *ArtifactAssetRelationship*: the multiplicity constraint within these elements has the meaning of determining the relationship instances, their targets, and their sources. The relationship instances are determined by instances of abstracts *Terms* referenced through *+gid* in the *+content* slot of the *m* constraint. The targets and sources of these relationship instances are all the instances of targets and sources from the abstract relationship that have direct reference to the referenced term instance or have a term that originates from it in their *Description* slot.

- *Claims* and further *ModelElements*: indicates the number of instances of claims and model elements is related to the number of instances of abstract *ExpressionElements* referenced inside its *Description* via *ExpressionLangString* elements.

The specification of *m* constraints should be consistent throughout the *ModelElements* of a SACM argument pattern specification. Consistency means that each abstract *ModelElement*, e.g., *Claim*, *AssertedRelationship*, referencing (directly or indirectly) to an abstract *Term* with a *m* constraint should also have a *multiplicity* constraint.

Example: consider the abstract *Term t* and a system design model describing all system hazards. For instantiating *t* for each system hazard it should have *multiplicity(m)* and a *mapping(p)* constraints. The EOL query *Hazard.all.collect(c | c.name)* is defined within *p* to get the names of all hazards from the external artifact defined *+externalReference* slot of *t*. As a result, the abstract *Term t* will be instantiated with the name of each system hazard.

3.4 OPTIONAL

Optional (o): can be attached to any SACM *ModelElement*, except *Terms* and *Expressions*. **Semantics**: *o* is used to denote *zero-or-one n-ary* cardinality of an abstract *ModelElement*, e.g., *Claim*, *AssertedRelationship*, in an argument pattern. The semantics of an *o ImplementationConstraint* is similar to GSN *Multiplicity* pattern extensions attached to a *GSN SupportedBy* element. *Optional* are used to assign Boolean conditions under *element property* (*p*) values (*v*) retrieved from an *external artefact*, that should be satisfied for instantiating an abstract *ModelElement*.

An element property *p* can assume different values *v*, e.g., v1 or v2. To achieve this, a *TaggedValue* element should be added to a *Term t* with *p* set as *+key* and the *+value* set to an *ExpressionLangString* with the *+expression* slot set to an abstract *Term t'*. Thus, the *+value* of the instances of *t'* will determine the value *v* of the key *p* for each instance of *t*. The *+taggedValue* of an *ModelElement* is all the element properties, i.e.,*+taggedValue*, of each *Term* within its *Description*. Therefore, if one of the *optional* constraints assigned to an instance of *ModelElement* is not fulfilled, the *ModelElement* should not be instantiated, i.e., the element is removed from the instantiated argument.

Example: consider the *optional(o) ImplementationConstraint* with the following *Boolean* condition *$tv(ENG, ASIL) = "ASIL C"* under the value of an instance of the abstract *Term componentX* attached to an abstract *Claim "{componentX} is ASIL C"* of an argument pattern. It is also necessary to consider that the abstract *Term componentX* contains a *TaggedValue* with its *key* slot set to "ASIL" and its value set to an *ExpressionLangString* referencing the abstract *Term ASIL*. This term should originate from *componentX* and have a constraint *mapping(p)* with a parameterized query. This

enables the correct association of ASIL *TaggedValues* within instances *componentX*. During the instantiation, the *optional* query *$tv(ENG, ASIL)* is replaced by the *+value* of the *TaggedValue* which has key *ASIL* and is related to instances of the abstract *Term componentX* within an English(ENG) multi lang *Description* of the *Claim*. If the *optional* condition is not satisfied for an instance of the *Claim* this instance is removed from the resulting model. Considering tree components MechanicPedal, ElectronicPedal, and BrakeUnit where MechanicPedal and ElectronicPedal are ASIL C, then, the instances generated for the *Claim* are "MechanicPedal is ASIL C" and "ElectronicPedal is ASIL C". The instance of the *Claim* "BrakeUnit is ASIL C" is deleted because it did not satisfy the condition *$tv(ENG, ASIL) = "ASIL C"* during instantiation.

3.5 CHOICE

Choice (c): constraint can be attached to *ArtifactAssetRelationship* and *AssertedRelationship* elements. **Semantics**: it can be used to denote possible alternatives (choices) in satisfying a relationship analogous to *GSN Optionality* extension (GSN, 2018). A *choice* constraint can be used to represent *1-of-n* or *m-of-n n-ary* selection of *source nodes* of an abstract SACM *AssertedRelationship* or *ArtifactAssetRelationship*. The *source* nodes of an *AssertedRelationship* can be *Claim* or *ArtefactReference* elements. *Artefact* elements (*Artefact*, *Activity*, *Event*, *Participant*, *Technique*, *Resource*) can be the *source* nodes of an *ArtifactAssetRelationship*. Choice constraint supports the specification of *lower* and *upper bounds* for the selection of *source* nodes of an SACM relationship.

The *choice* implementation constraints can be combined with *optional(o)* constraints to define conditions that should be satisfied for instantiating (selecting) each *source* node of an *ArtifactAssetRelationship* or *AssertedRelationship* relationships. While *optional* constraints define the conditions to instantiate source nodes of the relationships, the *choice* constraint verifies if source nodes of relationships were correctly instantiated considering the number of instances and the information provided by external artifacts (e.g., design, analysis, process models).

Example: Consider a top-level claim c with an *AssertedInference* relationship to tree other sub-claims. Each sub-claim must have an *optional(o)* implementation constraint defining conditions to instantiate them. Thus, a choice c *ImplementationConstraint* with bounds set to *1-of-3* selection can be attached to the *AssertedInference* relationship to denote that only one of the sub-claims must be instantiated.

3.6 CHILDREN

Children (s): This constraint subtype can be attached to abstract *Terms*. **Semantics:** It has similar semantics of the *mapping(p)* constraint, i.e., mapping abstract *Terms* to elements from an external artifact via model query. Children constraints are used

to indicate hierarchical relationships (i.e., parent and child) between instances of the same abstract *Term* retrieved from an external artifact. A children constraint obtains the child elements of each model element retrieved from an external artifact by executing the queries stored into a *mapping(p)* constraint, and into a *children (s)* constraint associated with the given abstract *Term t*. A children *ImplementationConstraint* manipulates the result of a mapping constraint. For this reason, a *children* constraint should be used in conjunction with a *mapping* constraint to enable the recursive instantiation of abstract *Terms* based on the hierarchical structure, e.g., the structure of FTA results. In the Children instantiation, the parent *Terms* are stored in memory as *+origins* of the instantiated *Term* enabling the hierarchical relationship among them to be used in the instantiation.

Example: consider an abstract *Term t*, and recursive hierarchical relationships between components (parent) and sub-components (child) in the design model (i.e., an external artifact), a query that captures hierarchical relationships between components and sub-components of a design model can be specified as *Component.all.selectOne(c| c.name='$parent').subcomponents.collect(s| s.name)*. This type of query is attached to the +content slot of *children (s) ImplementationConstraint* subtype. At instantiation time the $parent parameter is replaced at first by the values retrieved from the *mapping(p)* to get the root nodes children. Then $parent parameter is replaced recursively by each resulting value from the *children(s)* execution. The final result is the instantiation of the abstract *Term t* for each value resulting from the execution of both (*p*, and *s*) constraints. If only a *mapping* constraint with a recursive closure is used to instantiate the abstract term there will not be a hierarchical relationship among the *Term t* instances to be used in the instantiation of recursive argumentation structures.

3.7 USAGE OF SACM PATTERNS EXTENSIONS

This section contains practical examples of how the assurance case pattern extensions can be applied. The first scenario (Section **3.7.1**) describes an example of a **Mapping** constraint to instantiate a pattern based on a component model. The second scenario (Section **3.7.2**) describes the application of a **Mapping**, **Multiplicity**, **Choice**, and **Optional** to instantiate a functional assurance case pattern. Finally, the last scenario (Section **3.7.3**) describes the usage of the **Children** and **Mapping** constraints for instantiating a fault tree result.

3.7.1 Component Decomposition Argument Pattern

Figure 3.3 shows an Excerpt of the Component Decomposition Argument Pattern which is crucial for ensuring safety assurance in complex systems. This pattern helps in structuring safety arguments by breaking down the overall safety case into smaller, manageable components. This pattern contains the *Claim 1* with no constraints referencing

an abstract term *component* in its description. This claim is decomposed into the sub-claim *Claim 2* by an AssertedInferenceRelationship with no constraints referencing the abstract term *port*.

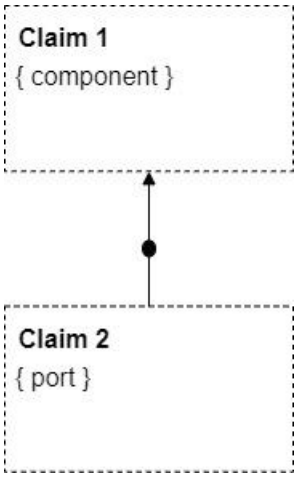

Figure 3.3 – Components Pattern Excerpt.

The abstract terms *component*, and *port* have a *mapping(p)* constraints sub-types (Table 3.1). The term *port* has its *+origin* property as *component*. The *component* term has a *mapping(p)* constraint with the query set to *Component.all.collect(c | c.name)*. The *port* term has a *mapping(p)* constraint with the query set to *Component.all.selectOne(c| c.name='$origin').ports.collect(p| p.name)*. These queries retrieve the components and ports names from the external model.

Table 3.1 – Components Pattern Constraints.

Type	**Name**	**Constraints**	**Origin**
Term	component	Mapping	-
Term	port	Mapping	component
Claim	Claim 1		
Claim	Claim 2		

Figure 3.4 – Components External Model.

Figure 3.4 contains an example of an external model. This model has one component "AddBrake" and its two ports "In" and "Out". Figure 3.5 shows the components pattern instance where the abstract term *component* was instantiated with the name of each system component retrieved from the external model generating an instance of *Claim*

1. The abstract term *port* is instantiated with all system ports related to an origin component generating an instance of *Claim 2*. In this example *Claim 1* is instantiated for the component "AddBrake" and *Claim 2* for its ports "In, Out".

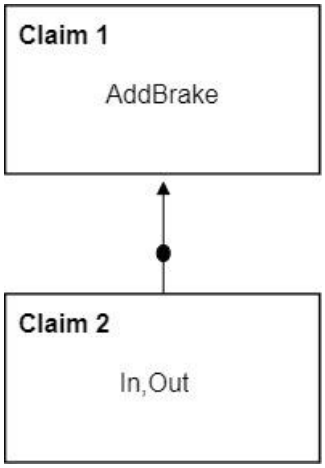

Figure 3.5 – Components Pattern Instance.

3.7.2 Functional Breakdown Argument Pattern

Figure 3.6 shows the Functional Breakdown Argument Pattern (KELLY; MCDERMID, 1997) excerpt in SACM notation. It is a structured approach used in assurance cases to demonstrate that a system meets its safety and dependability requirements.

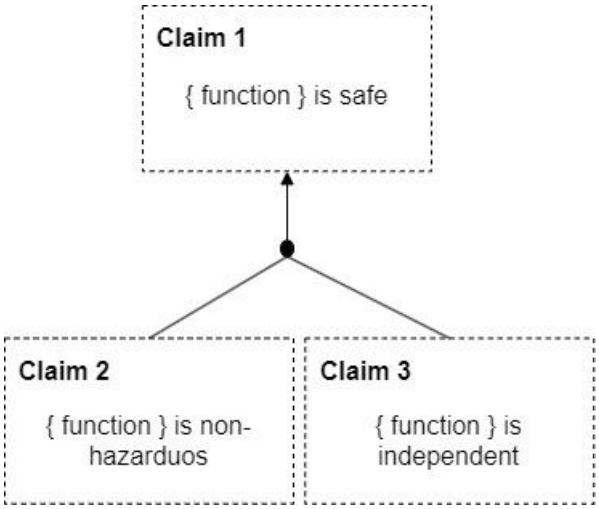

Figure 3.6 – Functional Breakdown Pattern Excerpt.

This pattern involves breaking down the overall safety argument into smaller, more manageable functional components, each with its own specific safety claims and evidence. In this structure, *Claim 1* is decomposed into two mutual exclusive selection sub-claims arguing two different sub-goals of functional safety: *Claim 2* argues that interactions between system functions are non-hazardous. *Claim 3* argues that all system functions are independent (no interactions). These claims have been extracted from the pattern in order

to illustrate an instantiation scenario. The reasoning behind the safety of the functions considering their type, i.e., non-hazardous or independent, are further decomposed into sub-claims in the real argumentation structure.

Table 3.2 describes each one of the main elements and their constraints. The abstract term *function* has a *multiplicity(m)* and *mapping(p)* constraints sub-types. This term is referenced within the description of *Claim 1*, *Claim 2*, and *Claim 3*. A *multiplicity(m)* constraint is attached to all of the claims due to the multiplicity of the abstract term referenced. A choice c *ImplementationConstraint* of *1-of-2* selection is attached to an AssertedInference element to denote that that only one *source* sub-claim between *Claim 2* and *Claim 3* should be present in one instance of Functional Breakdown argument pattern. Two optional constraints are present in the pattern specification. They define Boolean conditions under the function type of an instance of *function* abstract Term. If the function type is independent *Claim 3* should be instantiated. On the other hand, if the type is Non-hazardous *Claim 2* should be instantiated.

Table 3.2 – Functional Pattern Constraints.

Type	**Name**	**Constraints**
Term	function	Mapping, Multiplicity
Claim	Claim 1	Multiplicity
Claim	Claim 2	Multiplicity, Optional
Claim	Claim 3	Multiplicity, Optional
AssertedRelationship	-	Choice

Assuming that independent systems and non-hazardous systems do not contribute to the system hazard. The system hazards (Table 3.3) and the system functions (Table 3.4) are analyzed resulting in: the Anti-Icing System is independent, the Navigation Lights are non-hazardous, and the Communication System is hazardous and related to the hazard of Loss of Communication ASIL C.

Table 3.3 – Example of Identified Aircraft Hazards.

System	**Hazard**	**Asil**
Aircraft	Loss of Control	D
Aircraft	Loss of Communication	C
Aircraft	Loss of Navigation Capability	D

Table 3.4 – Example of Aircraft Functions

Function	**Type**	**Hazard**
Anti-Icing System	Independent	-
Navigation Lights	Non-Hazardous	-
Communication System	Hazardous	Loss of Communication

Figure 3.7 shows the *Claim 1* instantiated for each system function is independent or non-hazardous and their respective sub-claims related to the function type. In this example, an instance of *Claim 2* and *Claim 3* are produced with descriptions "Navigation light is non-hazardous" and "Anti-Icing System is independent" respectively.

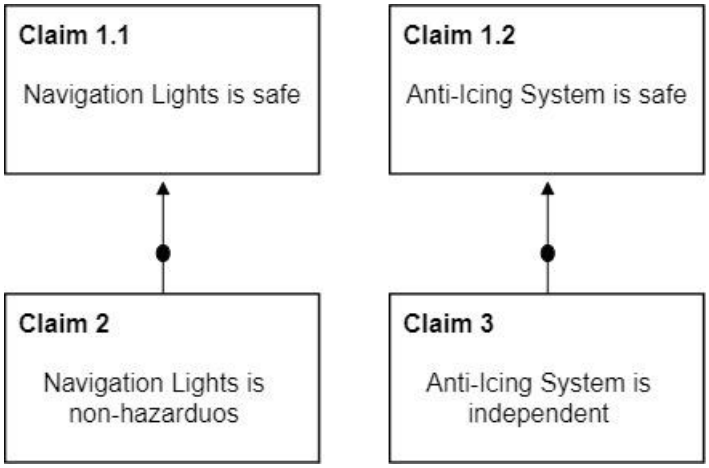

Figure 3.7 – Functional Breakdown Pattern Instance.

3.7.3 Hazardous Software Failure Mode Argument Pattern

The Hazardous Software Failure Mode Argument Pattern is a structured approach used in assurance cases to demonstrate that a software system meets its safety requirements by addressing potential hazardous failure modes (WEAVER, 2003). This pattern helps in systematically identifying, analyzing, and mitigating software-related hazards. This patterns aims to demonstrate that the occurrence of primary, secondary, and control failure modes of a given fault tree gate, e.g., AND/OR gates, do not lead the system to an unsafe state.

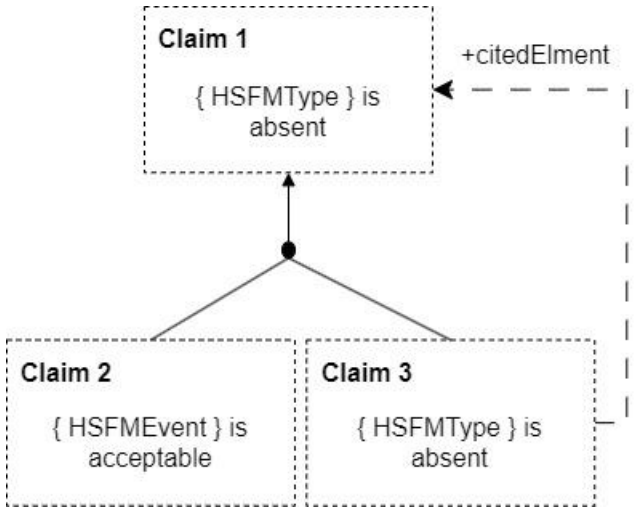

Figure 3.8 – HSFM Pattern Excerpt.

Figure 3.8 shows an excerpt of the Hazardous Software Failure Mode Argument

Pattern. The top-level *Claim 1* is decomposed into fault mitigation sub-claims *Claim 2* and *Claim 3*. *Claim 1* argues that all causes of each failure event specified in fault tree leaf nodes are acceptable, i.e., they do not lead the system to an unsafe state. *Claim 1* argues about the absence of an unsafe state through each non-leaf failure event of a fault tree node. *Claim 3* is a citation claim with its *+citedElement* property set to *Claim 1* as pattern reference for recursive instantiation.

Table 3.5 Shows the constraints within this pattern. The abstract term *HSFMType* has a *multiplicity(m)* constraint to allow the term to be instantiated multiple times. *HSFMType* also has a *mapping(p)* constraint mapping to top events of the fault tree and a *children(s)* constraint mapping to intermediate events of a fault tree. The *HSFMEvent* has origin in the *HSFMType* term and the constraints *mapping(p)* and *multiplicity(m)* mapping the term to all basic events of a fault tree related to a given top or intermediate event. The claims have *multiplicity(m)* constraints due the terms multiplicity.

Table 3.5 – HSFM Pattern Constraints.

Type	**Name**	**Constraints**	**Origin**
Term	HSFMType	Mapping, Multiplicity, Children	-
Term	HSFMEvent	Mapping, Multiplicity	HSFMType
Claim	Claim 1	Multiplicity	
Claim	Claim 2	Multiplicity	
Claim	Claim 3	Multiplicity	

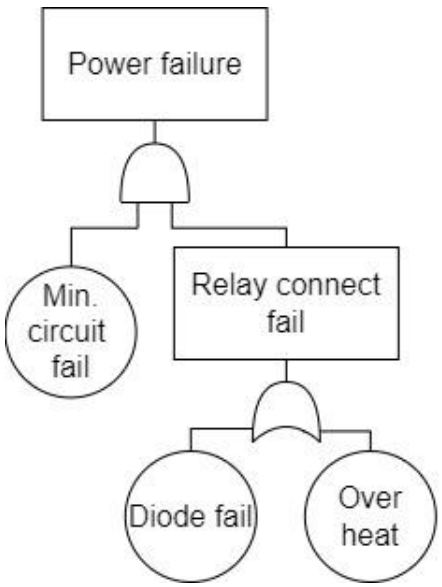

Figure 3.9 – HSFM External Model.

The external model (Figure 3.9) is a fault tree result generated after performing a safety analysis of the system. The path for this model is defined in the *+externalReference* property of the abstract terms *HSFMType* and *HSFMEvent*. The model is composed of

one top event "Power failure", one intermediate event "Relay connect fail", and three basic events, "Miniature circuit break fail", "Diode fail", and "Over heat".

Figure 3.10 shows the result of the instantiation. It starts the recursion instantiating *Claim 1* for the "Power failure" top event and *Claim 2* for the basic event "Miniature circuit break fail". In the next step *Claim 1* is instantiated for the "Relay connect fail" intermediate event because the citation property of *Claim 3*. Finally, *Claim 2* is instantiated for "Diode fail", and "Over heat" basic events.

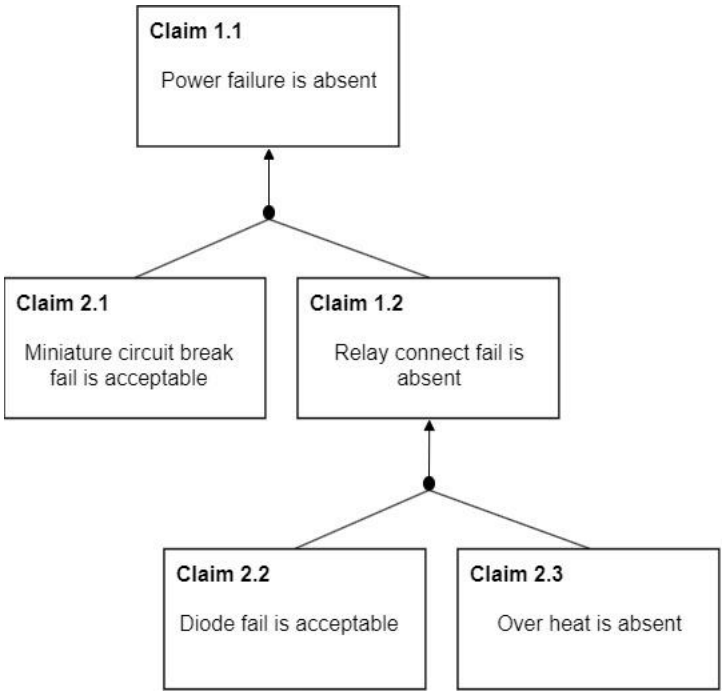

Figure 3.10 – HSFM Pattern Instance.

3.8 SUMMARY

This chapter presented the pattern extensions to the SACM metamodel and visual notation to support the specification and synthesis of executable SACM argument patterns with traceability links between claims and external artifacts. The proposed extension enabled the realization of the concept of Executable Digital Dependability Identity as another step toward system assurance at runtime. The extensions were proposed to add semantics for the ImplementationConstraint element from SACM to enable the specification of assurance case patterns with *Multiplicity*, *Optionallity*, and *Choice* present in GSN argument patterns (HABLI; KELLY, 2010). In addition, Mapping and Children SACM pattern extensions (i.e., *ImplementationConstraints*) have been proposed to enable the specification of executable argument patterns linked to evidence (fragments of model artifacts) in the SACM visual notation (SELVIANDRO; HAWKINS; HABLI, 2020). Those constraints can be attached to SACM abstract *Terms* to make the argument patterns executable, contributing to the synthesis, and modification of Assurance Case at runtime.

These constraints have been incorporated into SACM assurance case standard without any change in its meta-model, supporting the realization of the *Automated Assurance Case Instantiation* feature defined in SACM specification. To enable the specification of executable SACM assurance case templates enriched with the proposed extensions, the Assurance Case Editor, developed and integrated within the Eclipse Modeling Framework (EMF) platform is presented in the next chapter.

4 SACM ACEDITOR

SACM can be supported by model-based tools to fully exploit the benefit of automation brought by Model Based System Engineering (MBSE). This chapter describes a model-based tool for the specification of SACM assurance case and SACM assurance case patterns with support to SACM pattern extensions and automatic instantiation of assurance case patterns. Section 4.1 shows an overview of the ACEditor architecture and implementation. Section 4.2 presents the assurance case module. Section 4.3 describes the terminology module. Section 4.4 shows the artifact module. Section 4.5 presents the argumentation module. Finally, Section 4.6 describes the pattern extensions module.

4.1 ASSURANCE CASE EDITOR ARCHITECTURE

A previous work (NASCIMENTO, 2020) presented a preliminary version of the SACM ACEditor designed to support the specification and management of SACM assurance cases, and non-executable assurance case patterns. This chapter describes a new release, which supports the specification of assurance case patterns enriched with implementation constraints with explicit traceability links between the assurance case and evidence.

SACM ACEditor has been developed upon the Eclipse environment through the Graphical Editing Framework (GMF) (ECLIPSE, 2018b) for supporting the creation of the editor view, and features based on a SACM metamodel compliant with the EMF platform (ECLIPSE, 2018a). GMF enabled the development of graphical editors from EMF Ecore models. EMF is a widely used domain-specific modeling framework in the context of Model-Based Software Engineering (MBSE). The ACEditor supports the specification of custom implementation constraints sub-types. User-specific constraint sub-types can be defined by extending the Eclipse extension point *org.ufjf.sacm.aceditor.sacm2.ImplementationConstraint TypesProvider* (Figure 4.1).

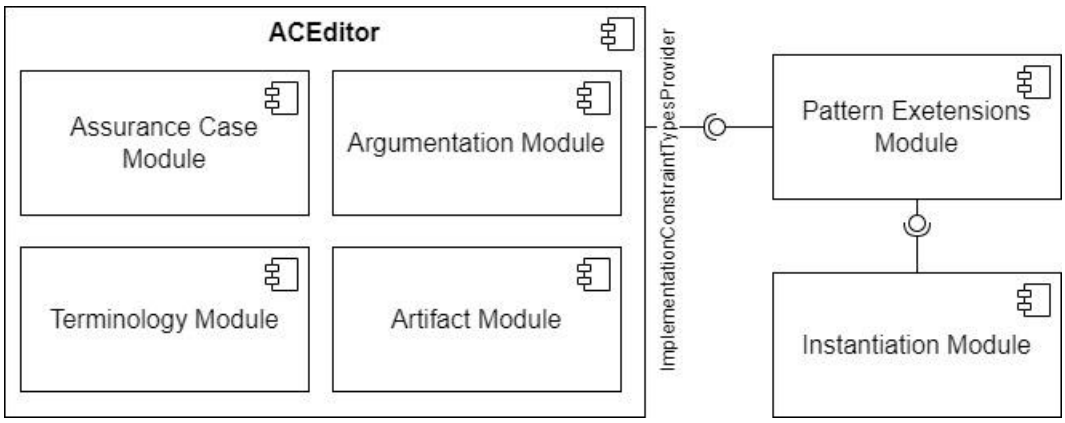

Figure 4.1 – SACM ACEditor Architecture.

The editor comprises the Assurance case, Terminology, Artifact, and Argumen-

tation modules. The Pattern Extension Module provides sub-types of constraints. The Instantiation Module is responsible for instantiating the pattern enriched with constraints. ACEditor provides extensibility, modularity, multi-language support, and traceability in the assurance case specification. The **extensibility** comes from the support for new sub-types of implementation constraints. This feature allows new types of constraints to be implemented with their own rules and graphical representation. With **modularity** assurance cases can be built using different package levels such as AssuranceCasePackages, ArgumentationPackages, ArtifactPackages, and TerminologyPackages. The **multi-language support** is achieved in assurance cases through the SACM MultiLangString element with one LangString for each desirable language. ACEditor allows the default language to be switched during the specification to show the descriptions of the elements in the chosen language (i.e. show only one LangString element from the multi-language Description). The **Traceability** between abstract terms and external information from design and analysis models is done through the definition of *+externalReference* property of abstract terms and the specification queries within the implementation constraints sub-types supported by the ACEditor tool.

4.2 ASSURANCE CASE MODULE

The Assurance Case Module implements the features of the SACM AssuranceCasePackage. It is the parent container for any assurance case in ACEditor and provides support for modular argumentation. The Assurance Case Module (Figure 4.2) contains a mixture of artifacts, argumentation, and terminology packages used for content exchange between users and packages. The specific types of packages supported by this module are the AssuranceCasePackages, ArtifactPackages, ArgumentPackages, and TerminologyPackages. The associative properties addressed are *+assuranceCasePackage*, *+interface*, *+artifactPackage*, *+terminologyPackage*, *+argumentPackage* of the AssuranceCasePackage meta-model element.

- *+assuranceCasePackage*: **AssuranceCasePackage** [0..*] (composition) – a collection of optional sub-packages;
- *+interface*: **AssuranceCasePackageInterface** [0..*] – a number of optional assurance case package interfaces that the current package may implement;
- *+artifactPackage*: **ArtifactPackage** [0..*] (composition) – a number of optional artifact sub-packages;
- *+terminologyPackage*: **TerminologyPackage** [0..*] (composition) – a number of optional terminology sub-packages;

- *+argumentPackage*: **ArgumentPackage**[0..*] (composition) – a number of optional argument packages.

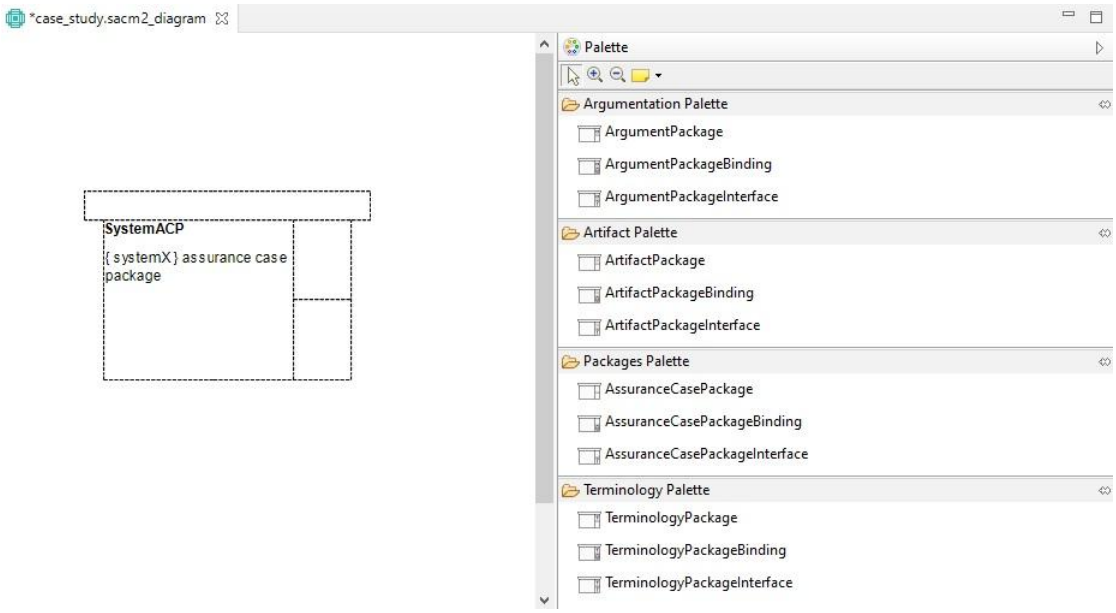

Figure 4.2 – Assurance Case Module View.

4.3 TERMINOLOGY MODULE

The Terminology Module of the ACEditor contains a set of features to support the terminology elements (vocabulary) specification and management. This module represents the TerminologyPackage of the SACM meta-model. It is the container element for any terminology assets. This module is composed of a set of TerminologyElements which can be Expressions, Terms, Categories, TerminologyGroups, or other TerminologyPackages including TerminologyPackageInterfaces and TerminologyPackageBindings addressed by the *+terminologyElement* property.

- *+terminologyElement*: **TerminologyElement**[0..*] (composition) – TerminologyElements contained in the TerminologyPackage, it can be either TerminologyPackage (and its sub-types) or TerminologyAssets (or its sub-types).

Figure 4.3 shows a table view of the Terms and Expressions of this module in a real assurance case pattern. It has been developed to improve the assurance cases management and specification. The terminology module also contains an implementation constraints overview table. It resumes all the implementation constraints sub-types assigned to the elements within the selected Terminology Module. It is useful for checking if they have been assigned to the correct Terminology Elements. Figure 4.4 shows an example of this table with abstract Terms and the types of implementation constraints assigned to them.

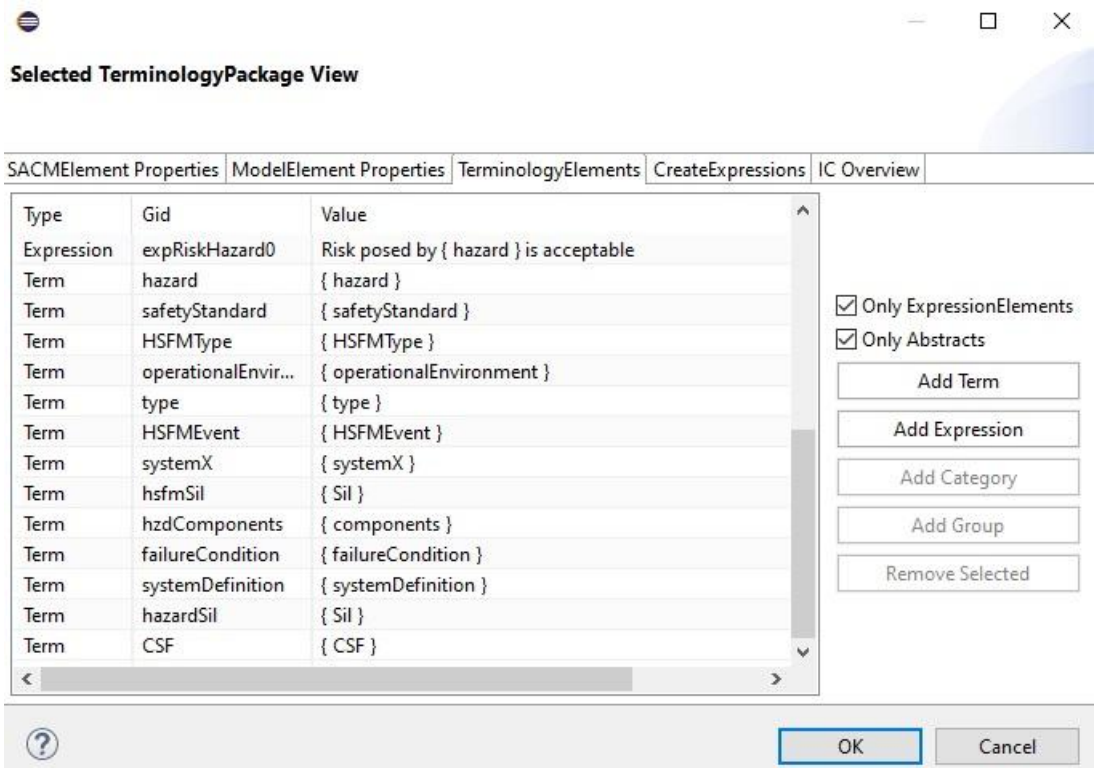

Figure 4.3 – Terminology Module View.

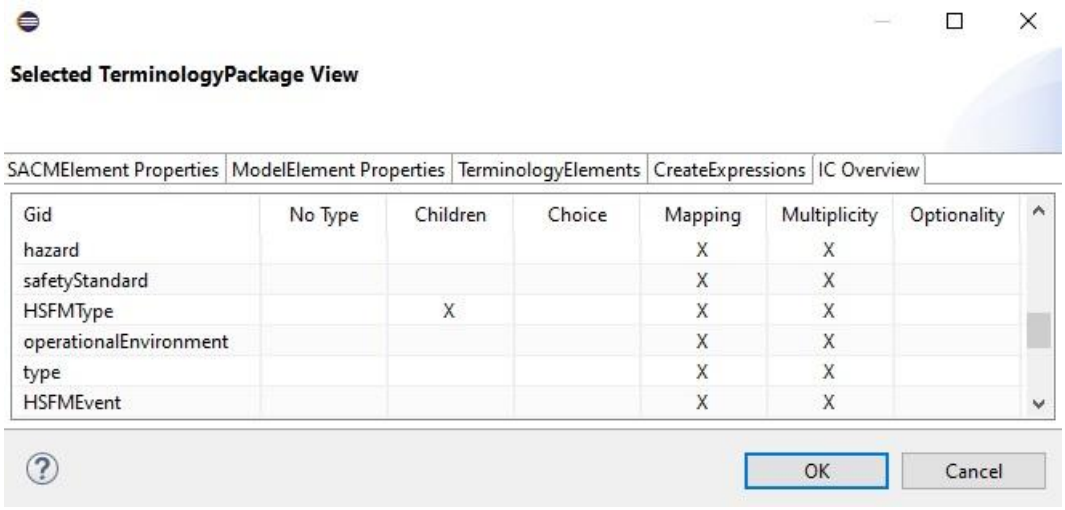

Figure 4.4 – Implementation Constraints Overview.

An assurance case vocabulary may have a large set of Terms and Expressions, thus, creating them can be time-consuming. The number of SACM properties and relations to be defined also difficult this process. Therefore, in order to address this problem, a two-step view for creating expressions from text has been implemented. The first view (Figure 4.5) has a text area for writing the text to be interpreted and translated into expressions. Natural language should be used to create the terms and expressions, and abstract terms must be written between brackets. A separator can be chosen for creating multiple expressions.

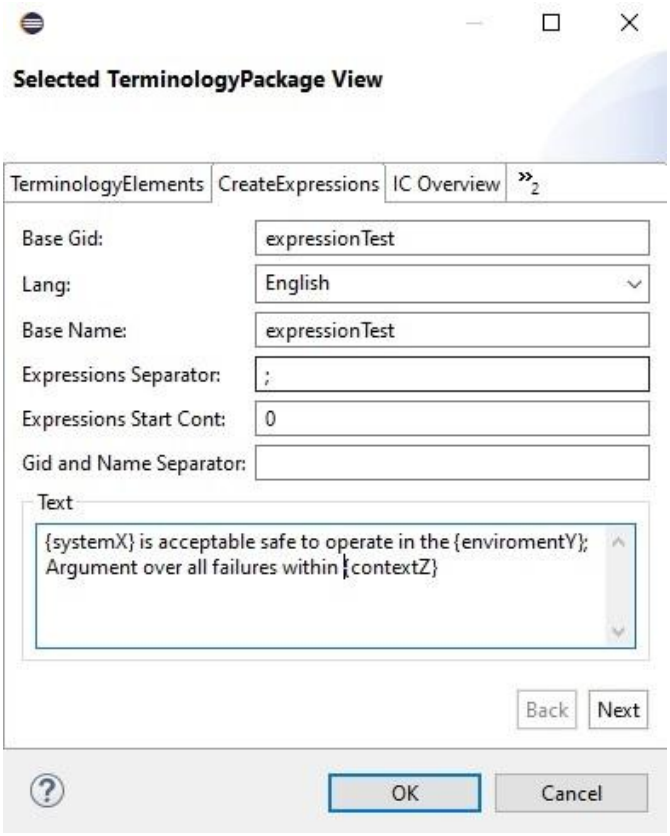

Figure 4.5 – Expressions Specification First Step.

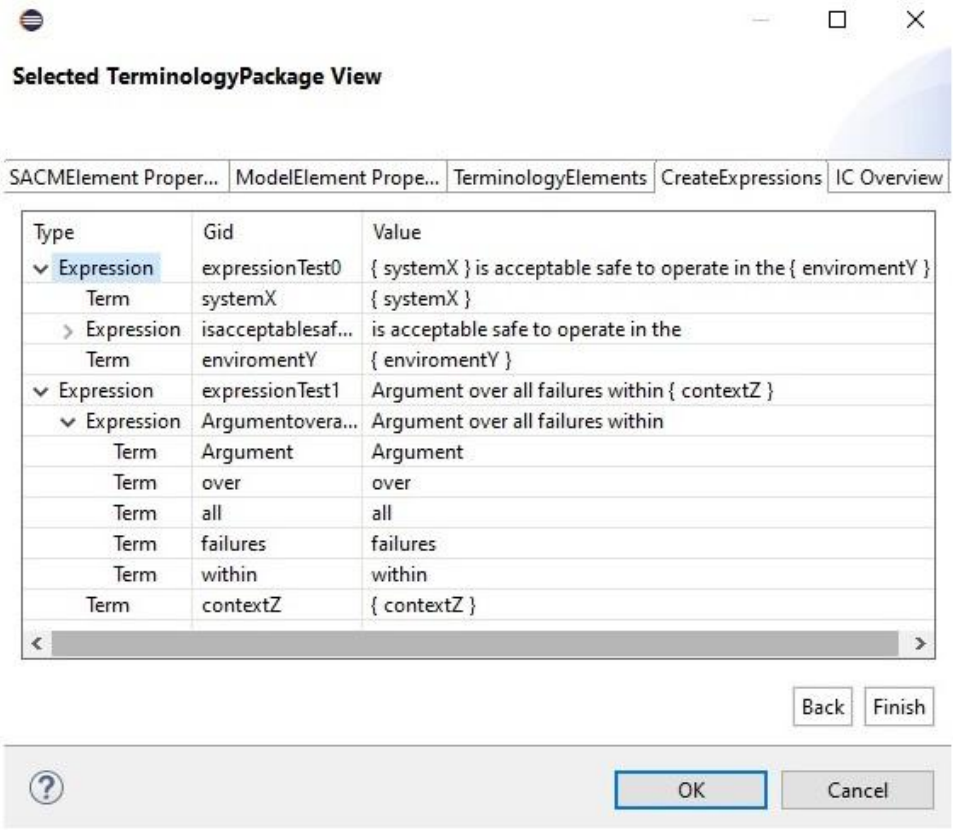

Figure 4.6 – Expressions Specification Second Step.

The second view for creating expressions (Figure 4.6) contains a tree table-like preview visualization of the expression elements, i.e., Terms and Expressions, that will be created. However, if one or more of them already exist in the model they are automatically loaded. The elements can also be edited in this view to assign other properties such as descriptions and implementation constraints. The output is one or more Expressions with the *+element* property referencing the terms informed in the text.

4.4 ARTIFACT MODULE

The Artifact Module of the ACEditor represents the ArtifactPackage of the SACM meta-model. It is a containing element for artifacts involved in a structured assurance case. It is composed of a set of ArtifactElements which can be Artifact, Activity, Event, Participant, Technique, Resource, ArtifactAssetRelationip, or other ArtifactPackages including ArtifactPackageIntefaces and ArtifactPackageBindings. Figure 4.7 shows a pattern example specified within the Artifact Module. It is possible to see that, aside from the elements within its domain, any other SACM element can be specified in this module. This is an SACM feature where not only external references are used as artifacts but assurance case themselves can also be used as artifacts to provide evidence for an argument structure. This is achieved through the *+artifactElement* property of the SACM ArtifactPackage.

- *+artifactElement*: **ArtifactElement**[0..*] (composition) – a collection of ArtifactElements forming an artifact package in a structured assurance case.

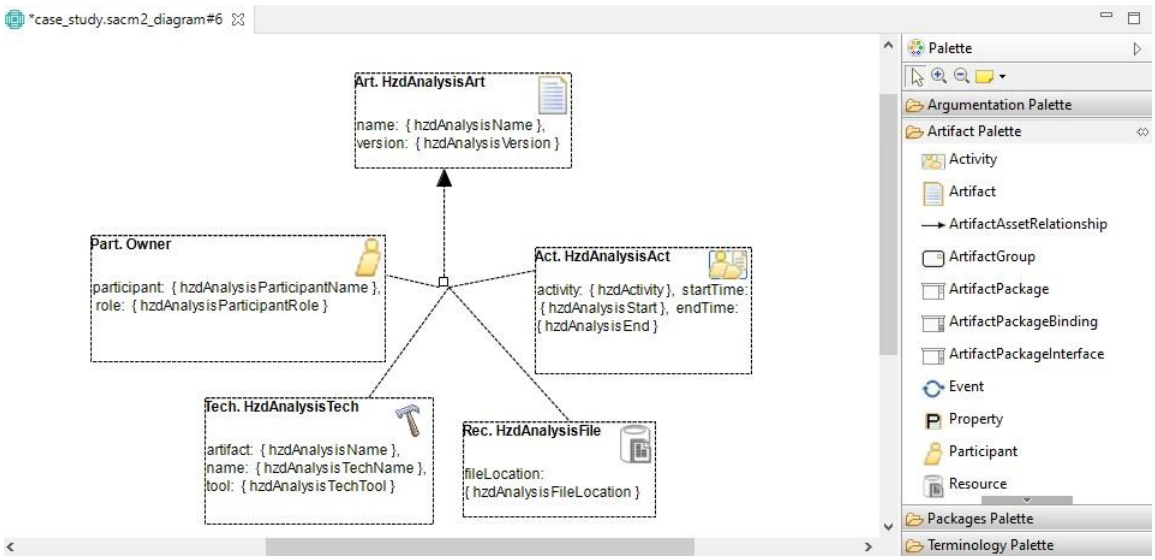

Figure 4.7 – Artifact Module View.

4.5 ARGUMENTATION MODULE

The Argumentation Module is one of the most important modules. In ACEditor it represents the SACM ArgumentPackage element. This module provides support for argumentation 'modules' within the assurance case. It is composed of a set of ArgumentationElements which can be Claim, ArgumentReasoning, AssertedContext, AssertedEvidence, AssertedInference, AssertedArtifactInference, AssertedArtifactSupport, or other ArgumentPackages including interfaces and bindings. These elements can be defined within the *+argumentationElement* property of an ArgumentPackage. Figure 4.8 shows the Hazard Avoidance argument pattern from (KELLY; MCDERMID, 1997) in SACM notation specified using the SACM ACEditor.

- *+argumentationElement*: **ArgumentationElement**[0..*] – an optional collection of ArgumentationElements organised within the ArgumentGroup.

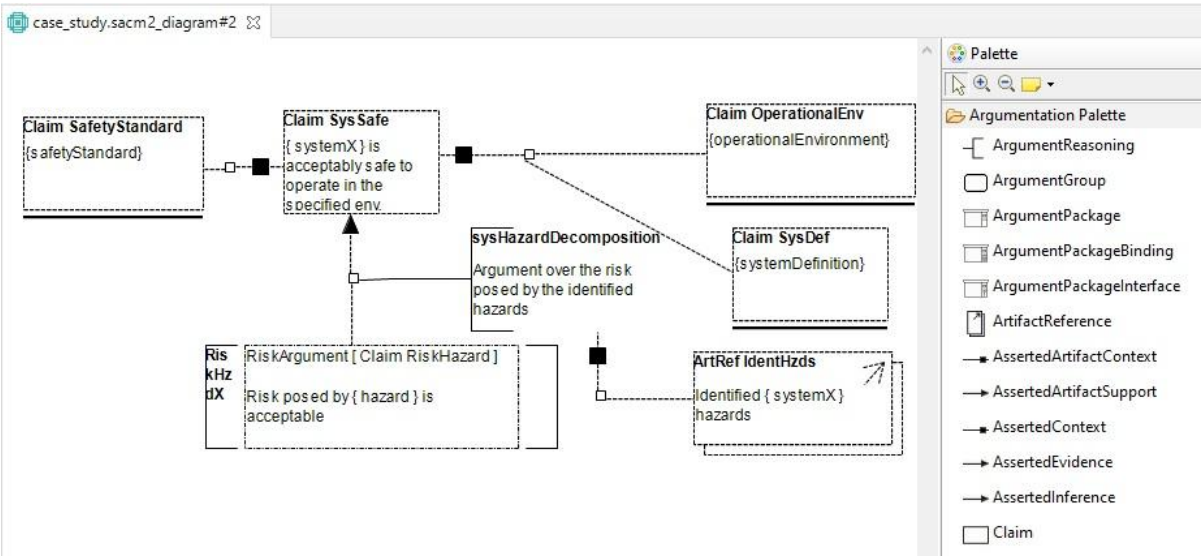

Figure 4.8 – Argument Module View.

4.6 PATTERN EXTENSIONS MODULE

The Pattern Extensions Module implements the ACEditor extension point for providing constraints sub-types. The types provided are the **Mapping**, **Children**, **Optionality**, **Multiplicity**, and **Choice** (Sections 3.2 to 3.6). Specific rules have been defined for each constraint sub-type using the extension point notation. These rules are dependent on the constraint sub-type domain, for instance, **Mapping** and **Children** constraints rules limit them to be applied only to abstract Terms. Views have been implemented in ACEditor to enhance the usability of the SACM assurance case pattern specification and implementation constraint sub-types. Specifically, there is a tab view for

editing ModelElement properties (Figure 4.9). This view allows the edition of the *+implementationConstraint* ModelElement property, thus, typed implementation constraints or natural language constraints can be added to the selected element. ACEditor also allows the edition of other ModelElement properties in this view, i.e., *+name*, *+description*, *+note*, *+taggedValue*. The list below describes the related properties of a ModelElement which can be edited in this view.

- *+name*: **LangString**[1] (composition) – the name of the ModelElement.
- *+implementationConstraint*: **ImplementationConstraint** [0..*] (composition) – a collection of implementation constraints.
- *+description*: **Description**[0..1] (composition) – the description of the ModelElement.
- *+note*: **Note**[0..*] (composition) – a collection of notes for the ModelElement.
- *+taggedValue*: **TaggedValue** [0..*] (composition) – a collection of TaggedValues, they can be used to describe additional features of a ModelElement

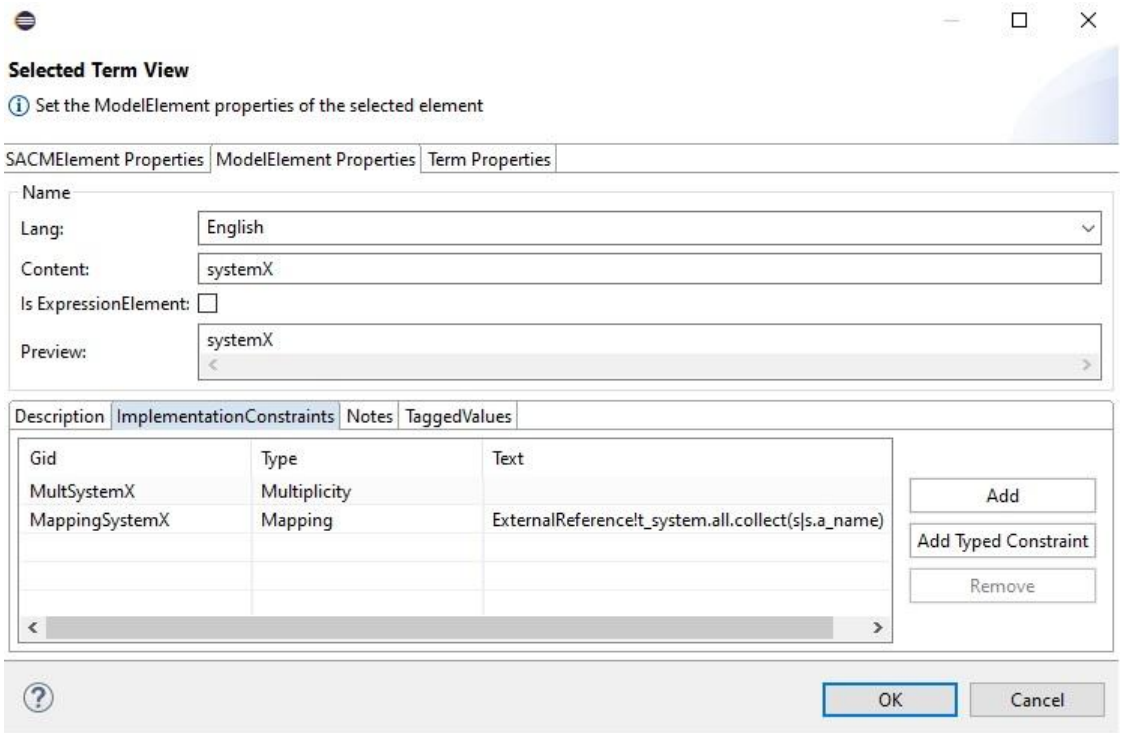

Figure 4.9 – Model Element View.

4.7 SUMMARY

This chapter presented an overview of the developed SACM ACEditor and architecture. The ACEditor has been developed within the Eclipse platform using GMF and

EMF for domain-specific modeling and model management. An extension point for the SACM ImplementationConstraint sub-types has been implemented. This capability was implemented to support the specification of pattern extension constraints **Mapping**, **Clidren**, **Multiplicity**, **Optional**, and **Choice**, described in Chapter 3, to SACM argument pattern elements. Next chapter presents a methodology to support the specification and instantiation of SACM assurance case patterns.

5 METHODOLOGY FOR SPECIFICATION AND INSTANTIATION OF SACM ASSURANCE CASE PATTERNS

This Chapter introduces a methodology to support the automatic generation of assurance cases synthesized from Fault Tree Analysis results. Section 5.1 provides an overview of the proposed methodology. Section 5.2 describes the Assurance Case Pattern Specification phase and Section 5.3 describes the Assurance Case Pattern Instantiation phase.

5.1 OVERVIEW

The proposed methodology is comprised of two major phases. The first phase, Assurance Case Pattern Specification, establishes the steps for creating an SACM assurance case pattern with traceability to external system artifacts including ODE-compliant Fault Tree Analysis results. The second phase, Assurance Case Pattern Instantiation, describes the steps required to generate FTA results and their transformation to one or more ODE models to enable the automated synthesis of executable SACM assurance case patterns.

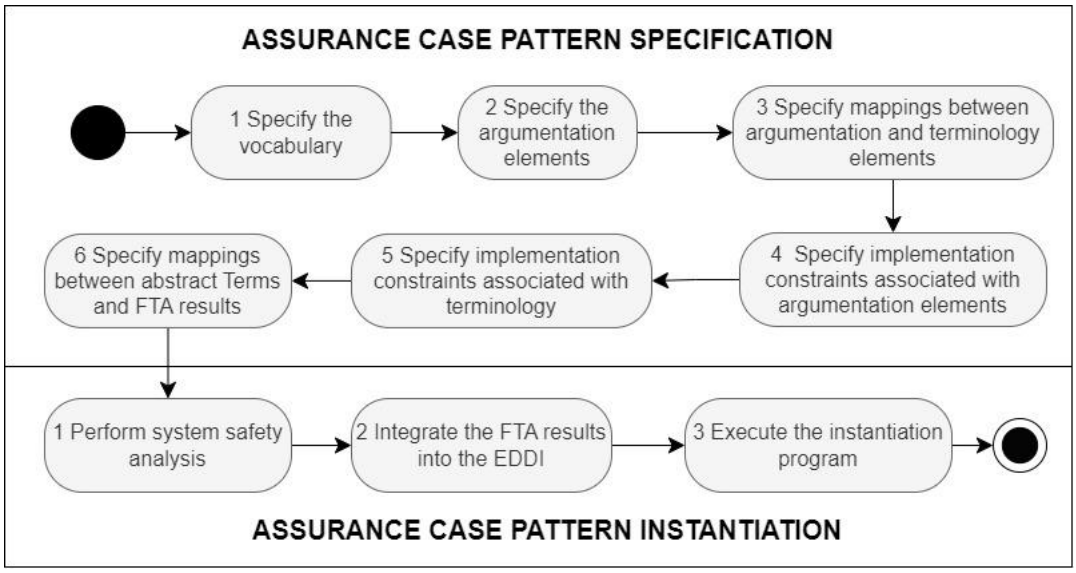

Figure 5.1 – Methodology Overview.

Figure 5.1 shows an overview of the proposed methodology. In the Assurance Case Pattern Specification phase, the vocabulary is specified followed by the specification of the argumentation elements and their mapping to the vocabulary within their Description. Next **Multiplicity**, **Optionality**, and **Choice** implementation constraints sub-types are specified for the argumentation elements then **Multiplicity** constraints are specified for the terminology elements. Finally, **Mapping** and **Children** constraints are specified for the vocabulary implementing the traceability to information within external artifacts elements through computer language queries. In the first step of the Assurance Case

Pattern Instantiation phase, the safety analysis is performed at system, function, and component levels. Then, in the second step, the tool-specific Fault Tree Analysis results are transformed to an ODE-compliant model which enables the execution of the automatic instantiation in the final step.

5.2 ASSURANCE CASE PATTERN SPECIFICATION

The inputs of this phase are the Hazard avoidance (KELLY; MCDERMID, 1997) and Hazardous Software Failure Mode (HSFM) (WEAVER, 2003) assurance case pattern catalog. In this phase, engineers specify the structure of the assurance case pattern using the SACM visual notation (OMG, 2021). SACM assurance case modeling tools such as ACME (WEI, R.; KELLY, T. P.; DAI, X., et al., 2019) and ACEditor (NASCIMENTO et al., 2023) can be used to support the specification of assurance case patterns with explicit links to evidence. This phase encompasses six steps detailed in the following:

Step 1: Specify the vocabulary. *Description*: In this step, we specify the placeholders using abstract SACM terminology elements (i.e., Terms and Expressions), and the textual information using concrete SACM Expression elements, which constitute the vocabulary of the targeted Hazard Avoidance and HSFM argumentation patterns used to build the assurance case pattern structure. Still in this step, we specify the relationships between abstract and concrete terminology elements to define expressions to be used as descriptions of SACM Claims, Reasoning, and ArtifactReference argumentation elements. Abstract SACM terms and expressions, via *+origin* and *+elements* properties respectively, provide the context to enable the automated instantiation of the pattern. SACM abstract Term and Expression elements enable an assurance case pattern specification to be machine-readable, i.e., it enables the specification of mappings between abstract terminology elements to the concrete information from design, safety assessment, and process models needed to instantiate abstract assurance case argumentation elements. *Output*: assurance case pattern vocabulary comprising concrete and abstract Term and Expression elements. Figure 5.2 shows all abstract terms and expressions produced in this step for the Hazard avoidance assurance case pattern specification. The *+origin* property of the terms *operationalEnvironment*, *safetyStandard*, *systemDefinition* and *hazard* are set to the abstract term *systemX*.

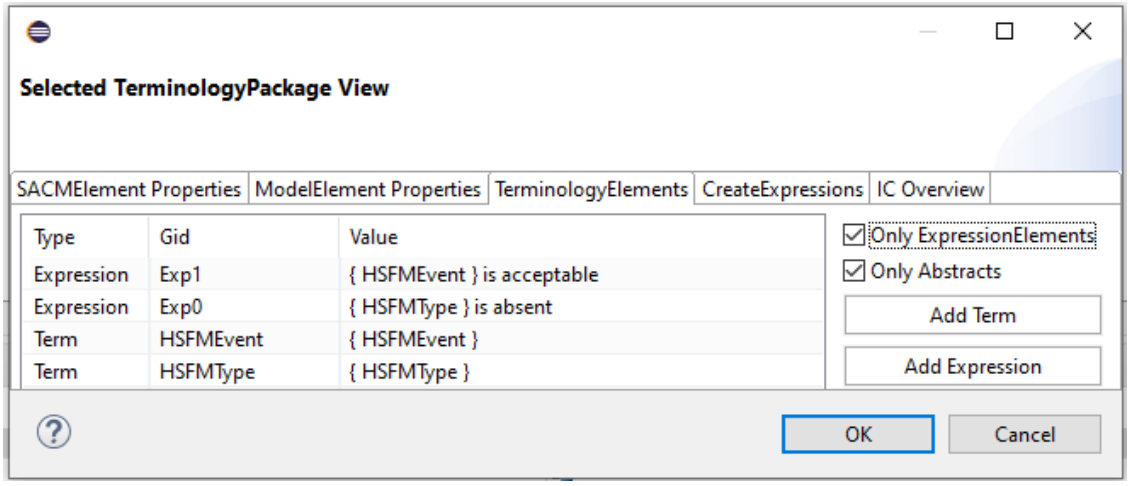

Figure 5.2 – Vocabulary Specification.

Step 2: Specify the argumentation elements. *Input*: the assurance case pattern vocabulary, i.e., abstract (non-instantiated) and concrete SACM terminology elements. *Description*: in this step, we specify the claims, artifact references, reasoning elements, and their relationships, using SACM AssertedRelationships, to define the hierarchical structure of the Hazard Avoidance and HSFM assurance case patterns. *Output*: the hierarchical structure of the pattern. Figure 5.3 shows an excerpt of the Hazardous Software Failure Mode assurance case pattern as a result of this step. The*Claim 1* is decomposed into two sub-claims *Claim 2* and *Claim 3*. *Claim 3* is a citation claim with its *+citedElement* property set to *Claim 1* as pattern reference for recursive instantiation. The claims and their characteristics are represented using the SACM visual notation.

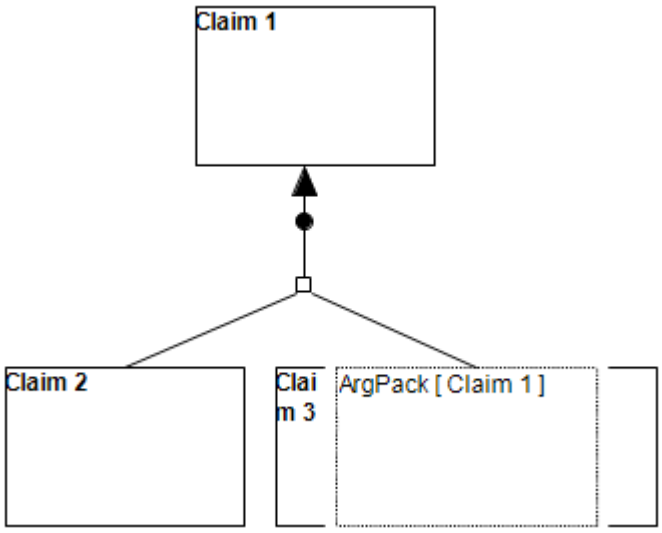

Figure 5.3 – Argumentation Elements Specification.

Step 3: Specify mappings between argumentation and terminology elements. *Input*: the assurance case pattern vocabulary and argumentation structure. *Description*: here, we define the description of each argumentation element (claim, reason-

ing, and artifact reference) specified in the pattern. A Description of an argumentation element includes explicit references to one or several SACM ExpressionElements. An expression element can be composed of both abstract and concrete SACM Term and Expression elements. *Output*: the assignment of descriptions to each SACM Claim, Reasoning, and Artifact Reference from the assurance case pattern. Figure 5.4 shows the result of this step where a Description had been assigned to the argumentation elements referencing vocabulary expressions created in **step 1**. This mapping has been done by adding English ExpressionLangString elements into the multi-language *+content* property of the argumentation elements Description. *Claim 1* argues that all causes of each failure event specified in fault tree leaf nodes are acceptable. *Claim 2* argues about the absence of an unsafe state through each non-leaf failure event of a fault tree.

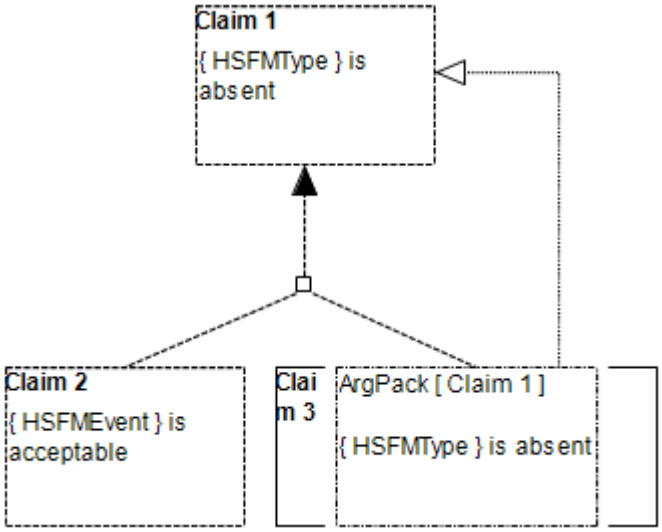

Figure 5.4 – Mapping Argumentation Elements to Vocabulary.

Step 4: Specify implementation constraints associated with argumentation elements. *Inputs*: the assurance case pattern vocabulary, structure, and SACM argumentation elements (Claim, Reasoning, and Artifact Reference) enriched with SACM Description elements. *Description*: here, we assign Multiplicity and Optional SACM Implementation Constraint subtypes, defined into the SACM pattern extensions (NASCIMENTO et al., 2023), to abstract Claims, Reasoning, and/or Artifact Reference assurance case pattern argumentation elements. In this step, we also assign SACM Choice implementation constraint to abstract SACM Asserted Relationship elements from the assurance case pattern. *Output*: assurance case pattern specification enriched with implementation constraints assigned to argumentation elements. Figure 5.5 shows the multiplicity constraints assigned to *Claim 1*, *Claim 2*, and for the *assertedInference* relationship.

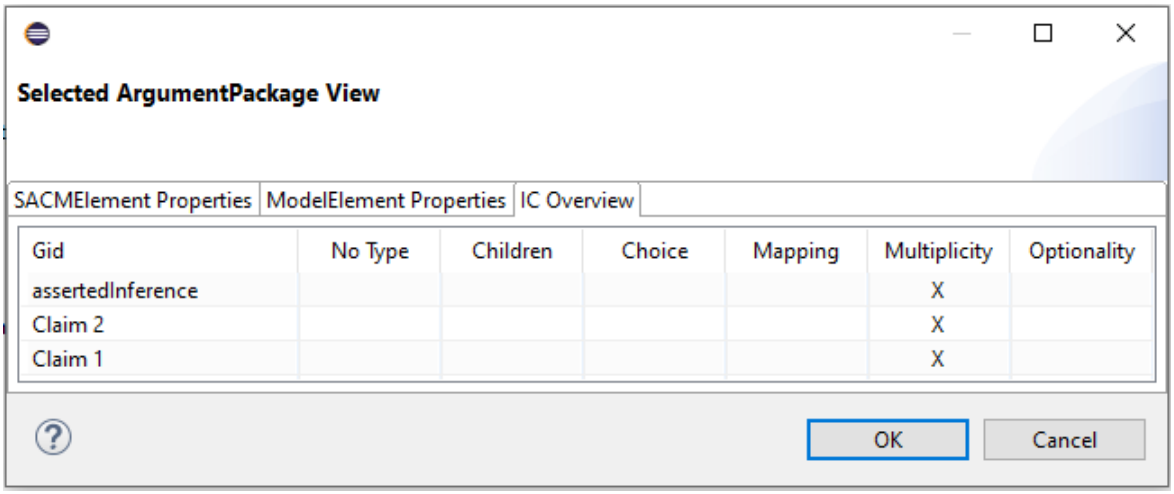

Figure 5.5 – Argumentation Elements Constraints.

Step 5: Specify implementation constraints associated with terminology. *Inputs*: the assurance case pattern vocabulary, structure, and SACM argumentation elements (Claim, Reasoning, and Artifact Reference) enriched with SACM Description and implementation constraints elements. *Description*: here, we assign Multiplicity, Mapping, and Children ImplementationConstraint subtypes, defined into the SACM pattern extensions, to abstract Terms. In this step, we also assign SACM Multiplicity implementation constraint to SACM Expression elements. *Output*: assurance case pattern specification enriched with implementation constraints assigned to terminology elements. Figure 5.5 shows the constraints associated with each terminology element. All the abstract terms have mapping constraints to retrieve their values from external artifacts and multiplicity constraints due to the number of tree events. The expressions that have referenced them through the *+element* property also have multiplicity constraints. The abstract term *HSFMType* has a children constraint to enable recursive instantiation of fault tree nodes.

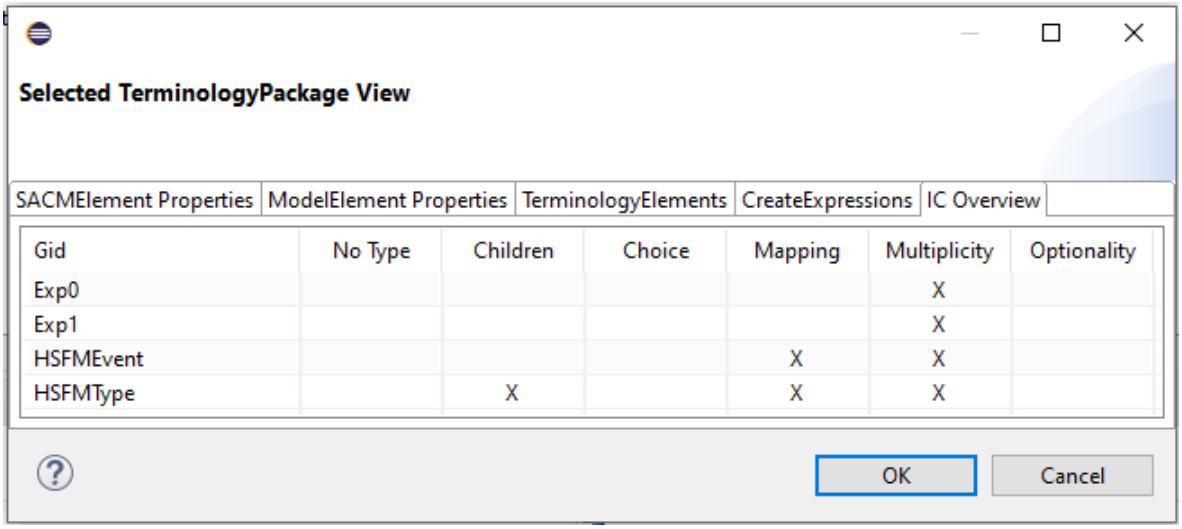

Figure 5.6 – Terminology Elements Constraints.

Step 6: Specify mappings between abstract Terms and FTA results. *Inputs*: the assurance case pattern vocabulary with implementation constraints, structure, and SACM argumentation elements (Claim, Reasoning, and Artifact Reference) enriched with SACM Description, implementation constraint elements, and the ODE metamodel. *Description*: in this step, we define model-based queries for the Mapping and Children implementation constraint subtypes assigned to abstract Terms. These queries provide traceability links between abstract terms of a pattern and ODE FTA package metamodel using a computer language such as EOL to map the values of these terms to FTA results. *Output*: an assurance case pattern specification enriched with traceability links to FTA results.

Listing 5.1 – HSFMType Mapping Query.

```
ExternalReference!Gate.all
    .select(c|c.description.contains("TOP-EVENT"))
    .collect(s|s.name)
```

Listing 5.1 shows query specified within the mapping constraint of the abstract term *HSFMType*. This query selects the names of all top events of the fault trees related to system hazards to instantiate the term. Listing 5.2 contains the children query of the *HSFMType*. This query recursively retrieves the values from fault tree intermediate events. It starts replacing the *$parentValue* parameter for the value of each top event and next for each value retrieved by its execution until an empty array is returned.

Listing 5.2 – HSFMType Children Query.

```
ExternalReference!Gate.all
    .selectOne(s|s.name="$parentValue")
    .causes.select(a|a.causeType.name = "Gate")
    .collect(c|c.name);
```

Listing 5.3 – HSFMEvent Mapping Query.

```
ExternalReference!Gate.all
    .selectOne(s|s.name="$originValue")
    .causes.select(a|a.causeType.name <> "Gate")
    .collect(c|c.name);
```

Listing 5.3 shows query specified within the mapping constraint of the abstract term *HSFMEvent*. This query selects the names of all basic events of the fault tree to instantiate it. The *HSFMEvent* has its *+origin* property set to the *HSFMType* abstract term. Thus, this mapping query is executed for each instance of *HSFMType*. The *$originValue* parameter is replaced with the *+value* property of its origin instances. Therefore, it returns

the names of all leaf nodes of a fault tree whose parent is a top event or an intermediate event.

5.3 ASSURANCE CASE PATTERN INSTANTIATION

This phase encompasses three steps that should be performed to generate an assurance case for a target system with references to FTA results.

Step 1: Performing system safety analysis. *Input*: system design. Description: in this step, the engineers must conduct safety analysis at both system, function, and component levels, e.g., using HAZOP at the system level to identify the potential hazards that malfunction the system, their safety risks, and safety goals; and fault tree analysis at function and component levels to identify how architectural subsystems and components may fail and contribute to the occurrence of hazards that may cause harm, and allocate functional and technical safety requirements to subsystems and components respectively. Model-Based Safety Assessment (MBSA) tools such as HiP-HOPS (PAPADOPOULOS et al., 2011), OSATE AADL (DELANGE; FEILER, 2014), CHESS framework (CHESS, n.d.), or EMFTA (CMU-SEI, n.d.) can be used to support this step. Outputs: fault trees describing the fault propagation paths for each identified system hazard. Figure 5.7 **A** shows a fault tree result as an example of this safety analysis step. The FTA result has one top-event called "Power failure", one intermediate event "Relay connect fail", and three basic events, "Miniature circuit fail", "Diode fail", and "Over heat".

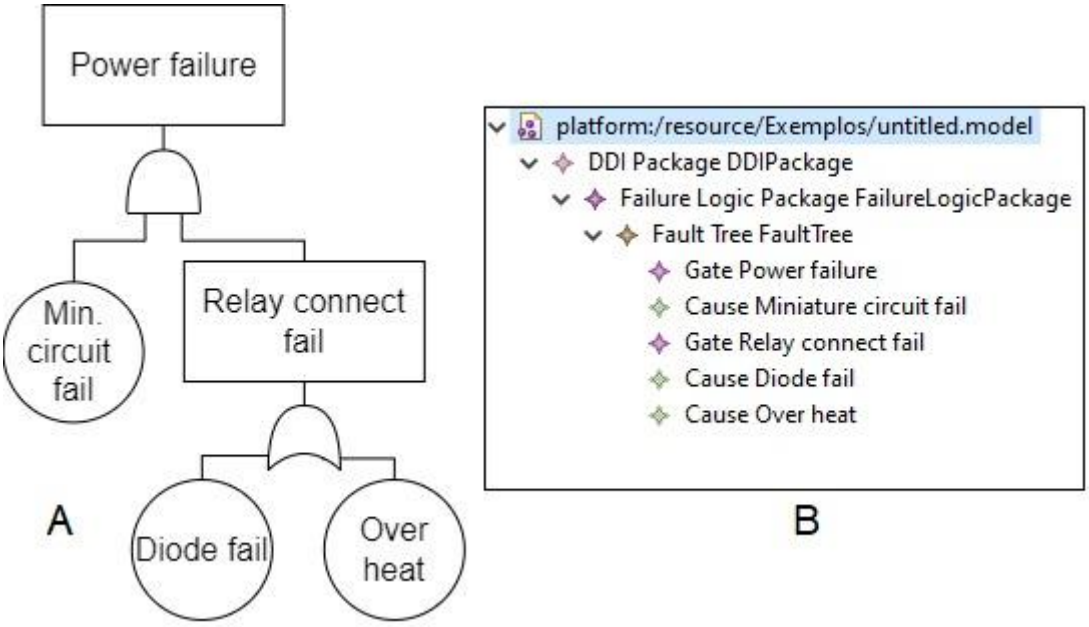

Figure 5.7 – Safety Analisys FTA Result / ODE Representation.

Step 2: Integrate the FTA results into the EDDI. *Inputs*: fault trees of each identified system hazard. Description: in this step, engineers execute a model

transformation algorithm developed to convert the input fault tree models, e.g., specified using HiP-HOPs, into the Open Dependability Exchange (ODE) metamodel compliant fault tree format. For fault tree models produced using third-part MBSA tools other than HiP-HOPs, e.g., Component Fault Trees (ZELLER et al., 2023) and OSATE AADL Error Annex (DELANGE; FEILER, 2014), the engineers need to specify a model transformation to map the tool metamodel elements to the ODE elements. The *ODE::FailureLogic::FTA Package* provides full support for representing FTAs, thus, the difficulty of this process lies in understanding the structure of the input fault tree and how it can be transformed to ODE. The Epsilon framework (ECLIPSE, 2022) offers languages and features for model management reducing the effort required for developing transformations from specific FTA models to ODE models. *Output*: ODE fault tree compliant models. Figure 5.7 **B** is the ODE-compliant fault tree result presented in the last step.

Step 3: Execute the instantiation program. *Inputs*: an assurance case pattern, and the ODE fault tree compliant models. In this step, engineers provide the pattern and the fault tree models to the assurance case pattern instantiation program, developed by the authors in (NASCIMENTO et al., 2023), to synthesize the system safety argument based on the FTA results. *Output*: a product safety argument for the target system with references to FTA results (evidence).

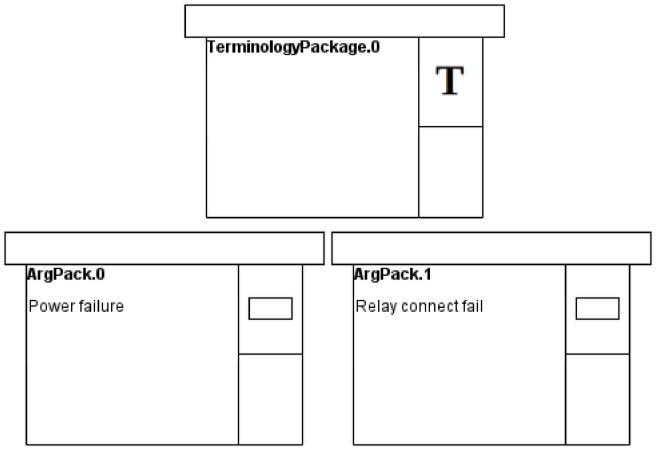

Figure 5.8 – Instantiation result package View.

The assurance case pattern is recursively instantiated resulting in two argumentation packages. One package contains the instantiated pattern for the Gate "Power failure" and the other for the Gate "Relay connect fail" (Figure 5.8). Figure 5.9 shows the instantiated pattern for the "Power failure" Gate. The top-level claim refers to a concrete instance of the term *HSFMType*. *Claim 2.0* has a reference to an instance of the *HSFMEvent* term generated for the "Miniature circuit fail" Cause. *Claim 3.0* is a citation claim referencing the top-level claim within the argument package generated from the pattern recursive instantiation.

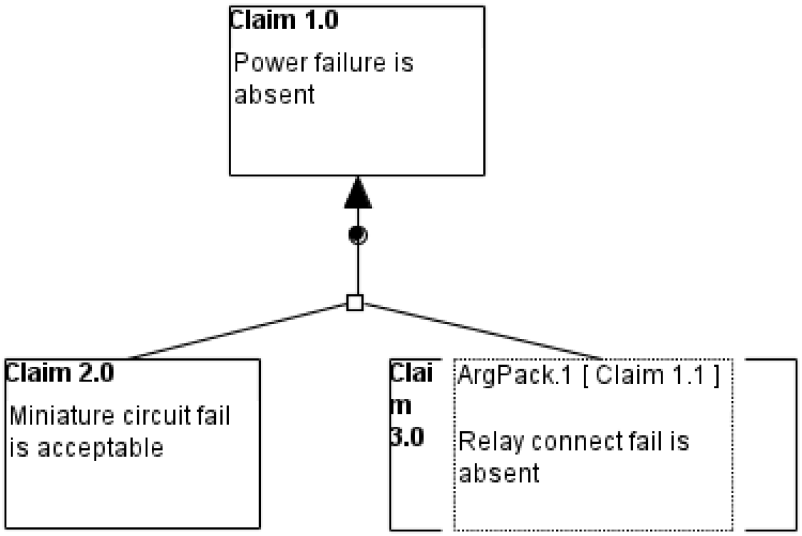

Figure 5.9 – Instantiation Result Power Failure ArgumentPackage View.

Figure 5.10 shows the argument package generated for the "Relay connect fail" Gate. The top-level claim refers to a concrete instance of the term *HSFMType* while *Claim 2.1* and *Claim 2.2* reference instances of the *HSFMEvent* term. *HSFMEvent* instances have been generated for the Causes "Diode fail" and "Over heat" related to the Gate "Relay connect fail".

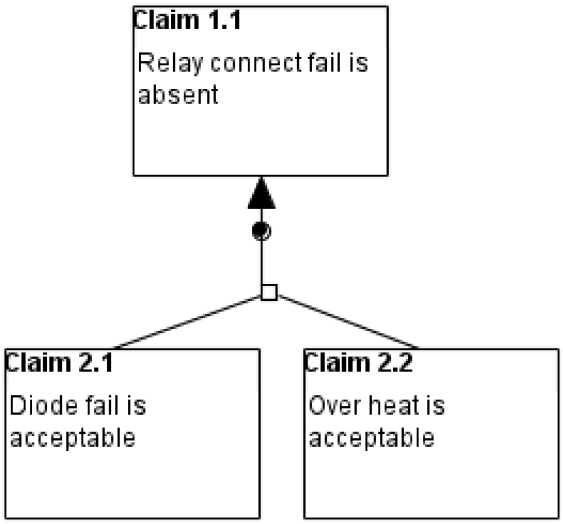

Figure 5.10 – Instantiation Result Relay Connect Fail ArgumentPackage View.

5.4 SUMMARY

This chapter presented a methodology to support the specification and synthesis of executable SACM argument patterns with traceability links between claims and ODE-compliant FTA results within an Executable Digital Dependability Identity. The methodology is comprised of two phases, Assurance Case Pattern Specification where

users specify the SACM assurance case pattern, and Assurance Case Pattern Instantiation where users perform safety analysis providing data for the instantiation of placeholders. Thus, executable assurance case patterns can be specified, mapping ODE-compliant FTA elements to abstract terms enabling automatic instantiation. The next chapter presents the instantiation algorithm to support the automatic instantiation of SACM assurance case patterns.

6 SACM PATTERN INSTANTIATION ALGORITHM

This chapter presents our model-driven approach for automated synthesis of executable SACM argument patterns from system design models. Section 6.1 provides an overview of the automated system of assurance case pattern instantiation algorithm. Section 6.2 covers the *EObject* module. Section 6.3 discusses the *ModelElement* module. Section 6.4 focuses on the *Package* module. Section 6.5 explores the *Term* module. Section 6.6 describes the *Expression* module. Section 6.7 details the *AssertedRelationship* module. Finally, Section 6.8 reviews the *ArtifactAssertedRelationship* module.

6.1 OVERVIEW

The instantiation algorithm was implemented using Epsilon EOL (KOLOVOS; ROSE, et al., 2013) and Java languages, and it executes on the Eclipse Modeling Framework (ECLIPSE, 2018a) platform. ***Input***: an executable SACM-compliant assurance case pattern specification model enriched with *ImplementationConstraints*, and MOF-compliant (e.g., EMF, UML) or other (e.g., Simulink, XML) design, analysis, and process models stated as *+externalReferences* of abstract *Terms* of the pattern. ***Output***: a product safety argument for the target system with references to external artifacts information. The proposed algorithm transforms a SACM assurance case pattern specification into a system assurance case. Thus, all the elements within the input model are generated in the output model replacing the placeholders defined in the pattern by system design, process, and analysis information. If an element has the *+isAbstract* property set to "true", it is instantiated in the output model. Otherwise, the element is copied to the output model.

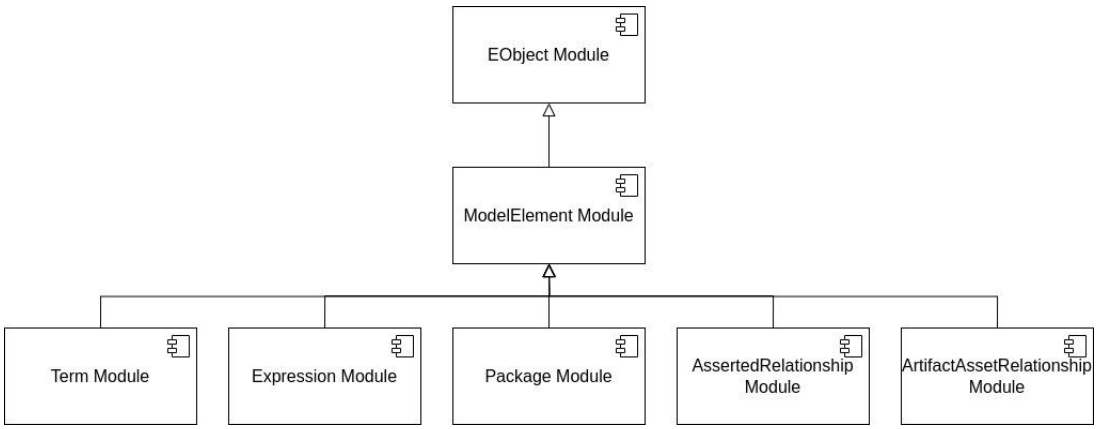

Figure 6.1 – Instantiation Modules.

Due to the large number of SACM meta-classes and different semantics, the implementation of the proposed assurance case pattern algorithm has been broken down into modules as illustrated in Figure 6.1. These modules are Java classes containing properties and operations to manipulate models through the Eclipse Epsilon and EMF

frameworks. Instances of these classes are created and SACM assurance case pattern elements are assigned to them in order to be instantiated.

The **EObject Module** defines the basic structure for instantiating a SACM element, and general concepts (e.g., copies management) used in the instantiation. This module is used for directly instantiating SACM LangString, Description, Note, and TaggedValue elements, and through inheritance by all the other modules. The **ModelElement Module** defines the basic structure for instantiating an SACM ModelElement. It is also used, through inheritance, by the Package, Term, Expression, AssertedRelationship, and ArtifactAssetRelationship modules. The **Package Module** is responsible for instantiating SACM AssuranceCasePackages, ArgumentPackages, TerminologyPackages, and ArtifactPackages and their bindings and interfaces. The **Term** and **Expression** modules contain the rules for instantiating abstract SACM Terms and SACM Expressions, respectively. Due to the difference in the number of targets, two modules have been created to support the instantiation of SACM asserted and artifact asset relationships. The **AssertedRelationship Module** supports the instantiation of SACM AssertedContext, AssertedEvidence, AssertedInference, AssertedArtifactContext, and AssertedArtifactSupport. AssertedRelationships can only have one element defined within their *+target* property. In contrast, ArtifactAssetRelationships can have multiple elements within their *+target* property. Thus, the **ArtifactAssetRelationship Module** provides rules for instantiating SACM ArtifactAssetRelationships among artifact elements.

Listing 6.1 provides an overview of the assurance case pattern instantiation algorithm. The proposed algorithm starts with the instantiation of abstract *term* enriched *mapping*, *multiplicity*, and *children* implementation constraints (lines 1-7). Firstly, for each abstract term *t*, it is verified if its *+origin* property has been defined (line 2). Next, if it is true, the mapping query within *t* is then executed for each instance of the *origin term ot*, replacing the "$originValue" query parameter for the value of the *origin term* (line 3). Therefore, instances of *t* are generated for all the values retrieved from the execution of the mapping constraint (line 4). If *t* has a children constraint, it is recursively instantiated (line 5). Thus, the children constraint query is recursively executed generating new instances of *t* (lines 6-7). For instance, the "$parentValue" query parameter of a children implementation constraint, e.g., the "$parentValue" of a *FTAGate.all.selectOne(g|g.name = "$parentValue").causes.collect(c|c.name) EOL query that returns all causes of a given hazard*, is replaced by the root values retrieved from the execution of a mapping constraint attached to *t*. Next, the query parameter is recursively replaced by the values retrieved from the execution of the query assigned to a children constraint until an empty list is returned. Later, model elements of a SACM assurance case pattern associated with each *abstract term* are instantiated by replacing references to *abstract terms* by references to their instances (lines 8-10).

Listing 6.1 – Overveiew of the instantiation algorithm.

```
01 for each abstract term t
02 | if t has an origin term(ot) instantiates ot
03 | executes the mapping queries p into the ODE model
04 | creates an instance of t for each value retrieved
05 | if t has a children(s) implementation constraint
06 | | for each instance t_i of t
07 | | | instantiates t recursively
   | | | executing s query on the ODE model
08 | for each model element(me) referring to t
09 | | for each instance t_i of t
10 | | | instantiates me replacing
   | | | the t reference by t_i,
   | | | e.g., "{SystemX}" by "MySystem"
11 for each relationship(r)
12 | creates a new instance of r
   | gouping the instantiated sources with
   | the related instantiated targets, i.e., they are linked
   | only if at least one instantied term within the source
   | had been direcly originated from the instantiated
   | terms of the target.
13 for each package(k)
14 | if k references a term(t)
   | trough multiplicity(m) constraint
15 | | for each instance t_i of t
16 | | | creates a new instance k_i of k
17 | | | put as content of k_i each instance me_i
   | | | of the model elements(me) within k
   | | | only if me_i has t_i or a term that directly
   | | | originated from t_i
18 | else
19 | | creates a new instance k_i of k
20 | | put as content of k_i all instances me_i
   | | of the model elements(me) within k
```

After instantiating model elements, each abstract SACM *AssertedRelationship* is instantiated based on its *target claim* (line 12). Thus, an instance of the *AssertedRelationship* is created for each instance of the target claim with its *+target* property set to its respective claim. The *source claims* are also instantiated, and defined within the *+sources*

property list of a concrete *AssertedRelationship* if they relate to the target claim. Therefore, instances of the source claims should be added to this list only if they contain at least one instantiated term, i.e., an abstract term t that becomes a concrete t_i, which is within the *Description* of the *target claim* or that has *originated* from *at least one term* within this *Description*. The algorithm ends with a set of steps to instantiate abstract SACM *packages* (lines 13-20). If the *package* has a multiplicity constraint, the *abstract term* referenced through its *+gid* property is instantiated (line 15). Then, for each one of its instances, a new instance of the *package* is created (line 16). The instantiated concrete term t_i related to a package instance determines its content. The *model elements* within a package are instantiated (line 17). However, only model elements with the same instantiated term (t_i) as the package instance, or at least one term that originates from t_i, are added to this package. On the other hand, if the *package* does not have a multiplicity constraint, then only one instance is generated (line 19). The content of this instance comprises all instances of model elements generated for the package (line 20). The instantiation modules implemented for the instantiation algorithm will be presented in the following sections.

6.2 EOBJECT MODULE

The **EObject** module provides the basic capabilities for instantiating SACM assurance case pattern elements (i.e., abstract and concrete terms and model elements). Those capabilities are reused across each level of the SACM assurance case pattern instantiation process. The ***inputs*** for this module are SACM LangString, Description, Note, and TaggedValue elements.

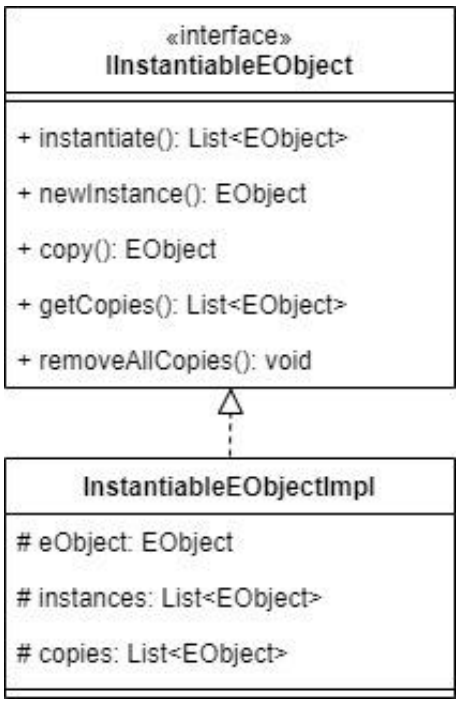

Figure 6.2 – Class Diagram EObject Module.

Figure 6.2 shows the EObject Module UML class diagram excerpt. The IInstantiableEobject interface includes a set of utility assurance case pattern instantiation operations that concrete classes should implement in each module of the proposed instantiation program. The InstantiableEObjectImpl implements the behavior specified in IInstantiableEObject for instantiating the SACM basic elements (i.e., elements that do not inherit from the SACM ModelElement meta-class) of an assurance case pattern specification. The InstantiableEObjectImpl class has *eObject* properties referencing the SACM pattern elements being instantiated. The SACM *pattern element instances* and their *copy* property list reference the instances generated within the product assurance case. The *instantiate* function returns a list of *instances*. The *newInstance* function generates instances of the *eObject* pattern element in the product assurance case adding it to the list of *instances*. The pattern element is instantiated with the *+isAbstract* property set to false, and the possible global identifier property (*+gid*) set to the pattern element *+gid* concatenated with the instance number. The EObject Module also defines a generic *copy* function that executes the *newInstance* function adding the generated element to the copies list (*copies*). The functions *getCopies* and *removeAllCopies* have been implemented to obtain and clear the copies list when necessary.

6.3 MODEL ELEMENT MODULE

The **ModelElement** module increments the basic capabilities of the **EObject** module allowing the instantiation of SACM ModelElement and their properties. These capabilities are reused across the Package, Term, Expression, AssertedRelationship, and ArtifactAssetRelationship Modules of the SACM assurance case pattern instantiation process. The inputs for instantiation in this module are SACM Claim, ArgumentGroup, ArtifactReference, ArtifactGroup, Property, Artifact, Activity, Event, Participant, Technique, Resource, TerminologyGroup, and Category elements. Figure 6.3 shows the ModelElement Module UML class diagram excerpt. The IInstantiableModelElement interface extends from the IInstantiableEobject and includes a set of utility operations that concrete classes should implement to enable the instantiation of SACM ModelElements. The InstantiableModelElementImpl implements the behavior specified in IInstantiableModelElement and extends from the InstantiableEObjectImpl. The InstantiableModelElementImpl class overrides the *newInstance* and *instantiate* functions of the InstantiableEObjectImpl. The *newInstance* method generates instances of the *eObject* pattern element in the product safety case without adding it to the list of instances. The pattern element is instantiated with the *+isAbstract* property set to false, and the possible global identifier property (*+gid*) set to the pattern element *+gid* concatenated with the instance number. Instances generated from *newInstance* method are only added to the list of instances within the *instantiate* function. This function returns the list of instances if the element has already been instantiated, otherwise, it executes the instantiation procedure adding instances to

the list and returning them at the end. This process depends on the model element being instantiated and its properties. Thus, *instantiate* method is overridden in the Package, Terms, Expression, AssertedRelationship, and ArtifactAssetRelationship modules.

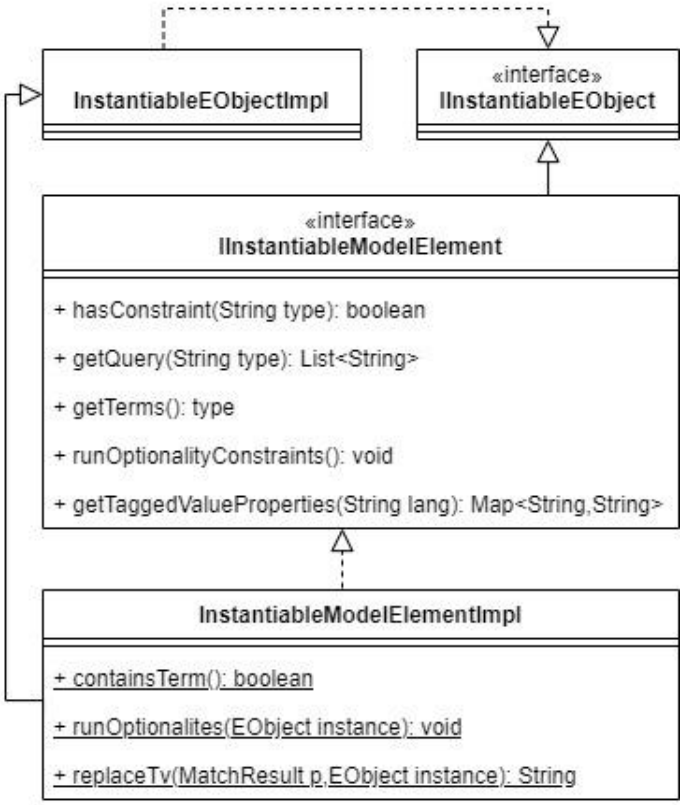

Figure 6.3 – Class Diagram ModelElement Module.

The InstantiableModelElementImpl class *instantation* method has two possible outcomes depending on the *+isCitation* and *+citedElement* properties of the ModelElement being instantiated. If the element is a citation (i.e., *+isCitation* property is true and *+citedElement* is not null) its instances are generated according to the *cited element instances*. Otherwise, if the element is not a citation its instances are generated according to its multi-language *+description* property. Thus, the ModelElement is instantiated based on a D set comprising its descriptions $\{d_1, d_2, ..., d_n\}$ where each d_i element can be either a LangString or an ExpressionLangString description. A matrix D' is defined where each D'_{ij} element is a specific instance j of a description i. Therefore, an instance of the ModelElement is generated for each column j of D' matrix, and its *+description* property is set to all descriptions instances J ($\forall i D'_{ij} \rightarrow J$).

The IInstantiableModelElement defines a few more methods that are implemented by the InstantiableModelElementImpl to enable the instantiation of SACM ModelElements. The *+hasConstraint* method verifies if the pattern model element has an implementation constraint sub-type. The computer language queries are obtained through the *+getQuery* method. The *+getTerms* function returns all the terms within the model element Descrip-

tion. The *+containsTerm* verifies if a model element has a specific term in its Description. After the instantiation of the model element, the *+runOptionalityConstraints* executes all optionality constraints in each instance. If an instance does not satisfy all these constraints it is removed from the product assurance case. The verification of the conditions is done by the *+runOptionalites*. This method runs all the optionalities queries for a specific instance. It uses the *+replaceTv* method for replacing the TaggedValue wildcards within optionality queries for its values in order to execute a boolean statement. For example, the wildcard "$tv(ENG, SIL) == A" checks if the model element has a TaggedValue with the lang set to "ENG", key "SIL", and value equal to "A". The TaggedValues are returned through the function *getTaggedValueProperties*. This function returns all the TaggedValues within the *+taggedValue* property and within the Expression/Terms elements referenced through ExpressionLangStrings by the Description of the model element.

6.4 PACKAGE MODULE

The **Package** module increments the basic capabilities of the **ModelElement** module allowing the instantiation of SACM Packages. The inputs for instantiation in this module are SACM AssuranceCasePackage, AssuranceCasePackageBinding, AssuranceCasePackageInterface, ArgumentPackage, ArgumentPackageInterface, ArgumentPackageBinding, ArtifactPackage, ArtifactPackageBinding, ArtifactPackageInterface, TerminologyPackage, TerminologyPackageBinding, and TerminologyPackageInterface elements.

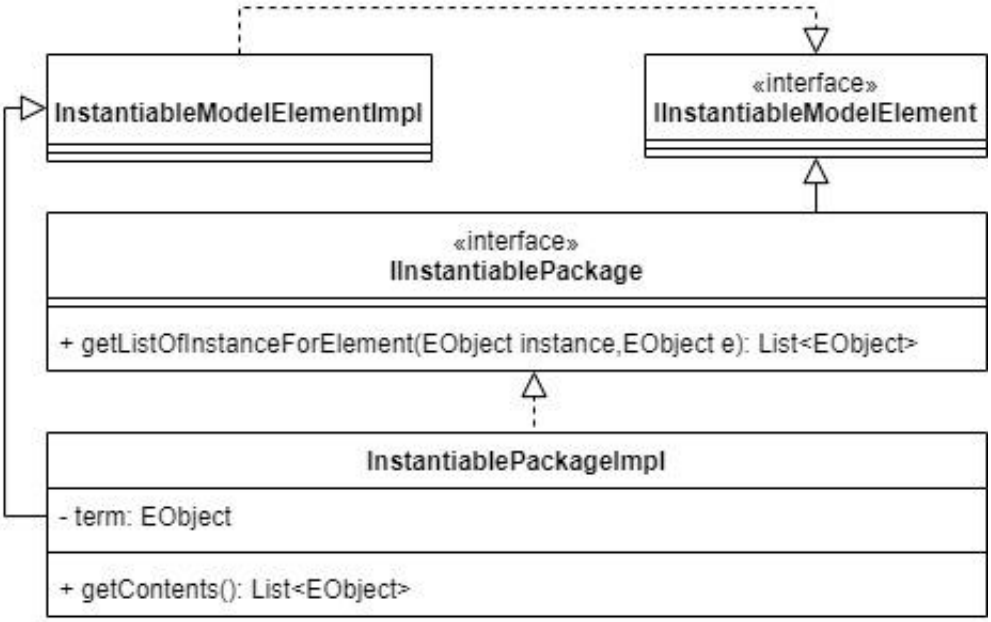

Figure 6.4 – Class Diagram Package Module.

Figure 6.4 shows the Package Module UML class diagram excerpt. The IInstantiablePackage interface extends from the IInstantiableModelElement and includes a set

of utility operations that concrete classes should implement to enable the instantiation of SACM Packages. The InstantiablePackageImpl implements the behavior specified in IInstantiablePackage and extends from the InstantiableModelElementImpl. The InstantiableEObjectImpl class has a *-term* property referencing or not a Term instance. The *instantiate* method has two possible outputs depending on the multiplicity constraint assigned to the package. If the package being instantiated does not have a multiplicity constraint, only one instance is created. Then all of its contents are instantiated and added as content of this instance. However, if the package being instantiated has a multiplicity constraint it is instantiated according to the Term referenced within this constraint by the *+gid* property. Thus, for each instance t_i of this term a new package instance is created with the *-term* set to t_i. The contents within the package are also instantiated and its instances are added as content of the respective package instance. Instances of elements instantiated from abstract elements that have a reference to a *-term* package instance term or another term that originates directly from it are added to the contents of this instance. Instantiated elements originated from pattern elements with *+isAbstract* property set to false are added as content off all the packages instances. These elements without placeholders are considered constants and should be present in the package independent of their *-term* property. The *+getListOfInstanceForElement* method returns the respective property list to add an element according to its type and the *+getContents* method returns all the package contents.

6.5 TERM MODULE

The **Term** module increments the basic capabilities of the **ModelElement** module allowing the instantiation of SACM Terms. The inputs for instantiation in this module are SACM Term elements. Figure 6.5 shows the Term Module UML class diagram excerpt. The IInstantiableTerm interface extends from the IInstantiableModelElement and includes a set of utility operations that concrete classes should implement. The InstantiableTermImpl implements the behavior specified in IInstantiableTerm and extends from the InstantiableModelElementImpl. The InstantiableTermImpl class has a *-origins* property list used for referencing the origins of a term. The methods *+addOrigin* and *+getOrigins* are used respectively for adding and removing elements of this *-origins* list. The *-wasAbstract* property is used to indicate if an instantiated Term has been instantiated from a Term with *+isAbstract* set to true or not. This flag allows the algorithm to return only the *Terms* instantiated from abstract *Terms* within elements (e.g. *Claim*) description, thus, the *-wasAbstract* property provides a way to relate these elements without considering pattern concrete *Terms*.

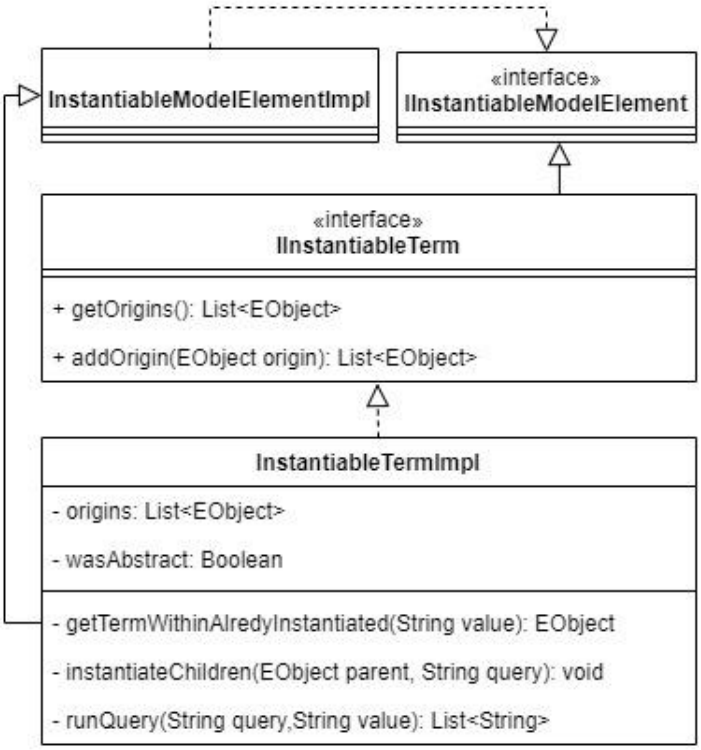

Figure 6.5 – Class Diagram Term Module.

The *instantiate* method of the InstantiableTermImpl generates an instance of a term according to its *+implementationConstraints* and *+origin* properties combination. An *abstract term* has a mapping constraint to define the *+value* property of its instance(s). If this *abstract term* has the *+origin* property defined, the mapping constraint query is executed for each origin instance, otherwise, only once. The values extracted from external models by the mapping queries are used to instantiate the *abstract term*. However, if this term has a multiplicity constraint an instance is generated for each value retrieved, if not, one instance is generated concatenating all these values. An *abstract term* with multiplicity constraint may also have a children constraint. Then, after the instantiation based on the mapping constraint for each instance t_i generated the recursive execution of children queries starts. Thus, for each value j retrieved by the children query execution a new t_{ij} is generated with *-origins* containing t_i.

To provide support for the instantiation the methods *-runQuery*, *-instantiateChildren*, *-getTermWithinAlredyInstantiated* have been implemented in the InstantiableTermImpl. The *-runQuery* method returns a list of values from the execution of a computer language query replacing the possible wildcards "$originValue" or "$parentValue" for their respective values (i.e., the *+value* of an instance Term from *+origin* or from the recursive instantiation). The *-instantiateChildren* method implements the recursive instantiation of terms based on the children constraint definition. The *getTermWithinAlredyInstantiated* method returns an instance of the instantiated term with a specific *+value* property. This method is used to prevent multiple instantiations of a term with the same *+value* property.

6.6 EXPRESSION MODULE

The **Expression** module increments the basic capabilities of the **ModelElement** module allowing the instantiation of SACM Expressions. The inputs for instantiation in this module are SACM Expression elements. Figure 6.6 shows the Expression Module UML class diagram excerpt. The IInstantiableExpression interface extends from the IInstantiableModelElement and includes a set of utility operations that concrete classes should implement. The InstantiableExpressionImpl implements the behavior specified in IInstantiableExpression and extends from the InstantiableModelElementImpl.

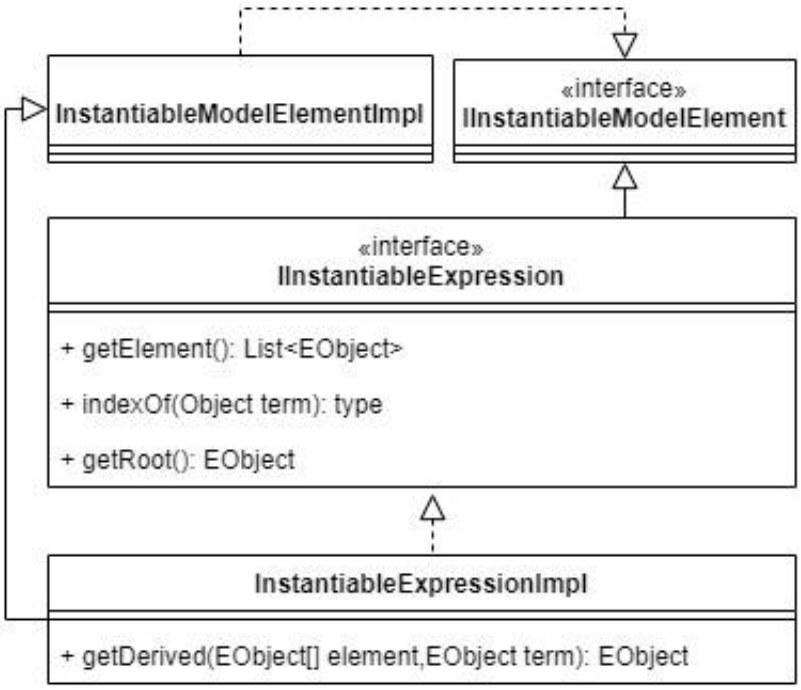

Figure 6.6 – Class Diagram Expression Module.

The InstantiableExpressionImpl *+getElement* method returns the *+element* property of the expression being instantiated. The *+getRoot* returns the first abstract term with no origin defined within the *+element* property of the expression. The *+getDerived* method returns the first abstract term that originates from a given abstract term within the *+element* property. The *indexOf* method returns the respective index of a term in the *+element* property. The *instantiated* method is implemented proving support for the instantiation of Expressions. At first, an array of *+element lists* is generated for each instance of the root term returned from the function *+getRoot*. Then, for each *+element list* of the array, the references to the abstract root term are replaced with references to each one of its instances. However, SACM expressions may have multiple abstract terms within the *+element* property thus the other possible abstract terms must also be instantiated. A top-down approach has been adopted to instantiate these terms. Thus, the *+getDerived* function is executed by passing the root term and returning the first

derived term within the *+element* list. Then, references to the derived abstract terms are replaced with references to instances in all *+element lists*. If a derived abstract term has multiple instances, a new *+element list* is generated for each instance replacing its reference. The final step of the instantiation is to generate expression instances according to these *+element lists*. If the expression has a multiplicity constraint an instance is generated for each one of these *+element lists*. However, if the expression does not have a multiplicity constraint only one instance is generated containing all these *+element lists* within the *+element* property.

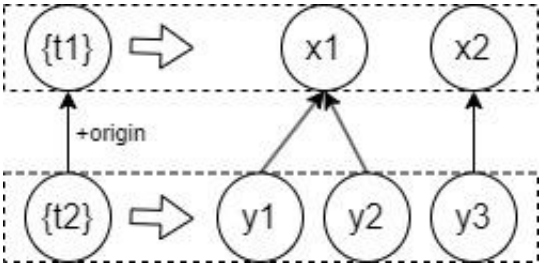

Figure 6.7 – Expression Terms.

Figure 6.7 shows a representation of this instantiation phase considering the abstract terms and their instances. The abstract term *t2* originates from the abstract term *t1* and both have multiplicity and mapping constraints. The term *t1* is instantiated in the terms *x1* and *x2*. The term *t2* is instantiated in the terms *y1* and *y2* originating from *x1*, and *y3* originating from *x2*.

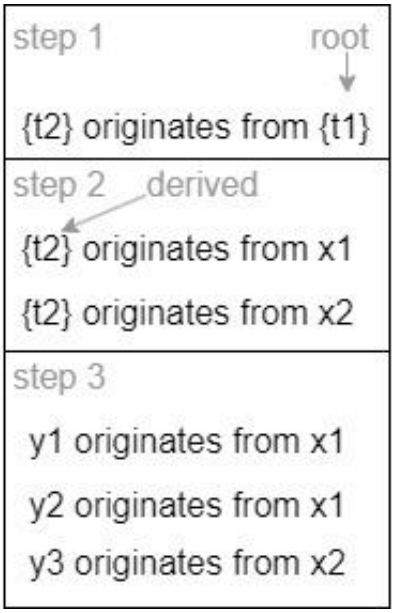

Figure 6.8 – Expression Instantiation.

Considering the abstract expression "{$t2$} originates from {$t1$}" (Figure 6.8). At step 1 the root element is selected and instantiated. At step 2 the derived term is selected

and then instantiated. This process continues until there is no abstract term to be instantiated in the list. Element lists are generated for all paths from the root to each leaf node resulting in step 3 with the lists "$y1$ originates from $x1$", "$y2$ originates from $x1$", and "$y3$ originates from $x2$".

6.7 ASSERTED RELATIONSHIP MODULE

The **AssertedRelationship** module increments the basic capabilities of the **ModelElement** module allowing the instantiation of SACM AssertedRelationships. The inputs for instantiation in this module are SACM AssertedContext, AssertedEvidence, AssertedInference, AssertedArtifactContext, and AssertedArtifactSupport relationship elements. Figure 6.9 shows the AssertedRelationship Module UML class diagram excerpt. The IInstantiableRelationship interface extends from the IInstantiableModelElement and includes a set of utility operations that concrete classes should implement to enable the instantiation of SACM AssertedRelationships. The InstantiableAssertedRelationshipImpl implements the behavior specified in IInstantiableRelationship and extends from the InstantiableModelElementImpl.

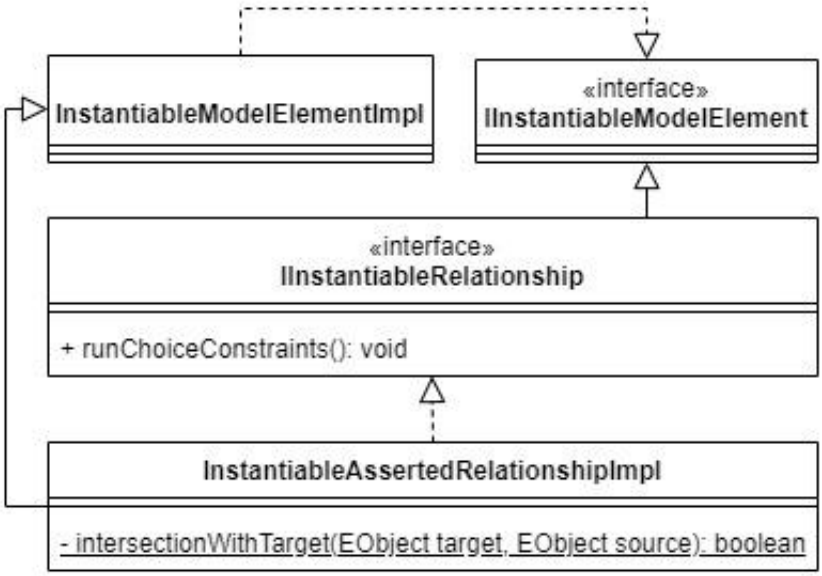

Figure 6.9 – Class Diagram AssertedRelationship Module.

The *instantiate* method of the InstantiableAssertedRelationshipImpl class generates an instance r_i of a pattern relationship(r) for each instance t_i generated for the element defined in r *+target* property. Then the elements within r *+source* property are instantiated. The method *-intersectionWithTarget* verifies if a given a source instance s_j contain at least one instantiated term, i.e., an abstract term that becomes a concrete, which is within the *Description* of the *target claim* t_i or that has *originated* from *at least one term* within t_i *Description*. If this condition is verified, s_j is added in the *+source* property list of r_i.

Thus, each instance of the relationship must have on its *+source* property elements that are related to the *+target* or constant elements (i.e., elements that do not have abstract terms). The method *runChoiceConstraints* is executed after instantiation on each instance r_i generated for the relationship. This method verifies if r_i attends the lower and upper bound of sources defined in the choice constraint of the pattern relationship r.

6.8 ARTIFACT ASSET RELATIONSHIP MODULE

The **ArtifactAssetRelationship** module increments the basic capabilities of the **ModelElement** module allowing the instantiation of SACM ArtifactAssetRelationships. The inputs for instantiation in this module are SACM ArtifactAssetRelationhip elements. Figure 6.10 shows the ArtifactAssetRelationship Module UML class diagram excerpt. The IInstantiableRelationship interface extends from the IInstantiableModelElement and includes a set of utility operations that concrete classes should implement to enable the instantiation of SACM ArtifactAssetRelationships. The InstantiableArtifactAssetRelationshipImpl implements the behavior specified in IInstantiableRelationship and extends from the InstantiableModelElementImpl.

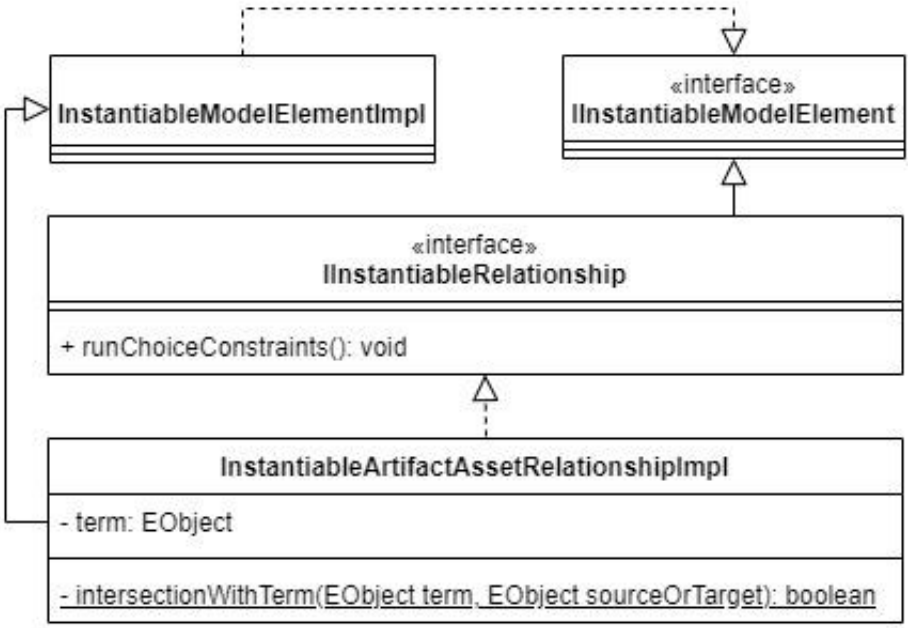

Figure 6.10 – Class Diagram ArtifactAssetRelationship Module.

The *instantiate* method of the InstantiableArtifactAssetRelationshipImpl class has two possible outputs depending on the multiplicity constraint assigned to the ArtifactAssetRelationship. If the relationship r being instantiated does not have a multiplicity constraint, only one instance r_i is created. Then all elements within its *+target* and *+source* properties list are instantiated and added to the respective property list of r_i. However, if the relationship r being instantiated has a multiplicity constraint, it is instantiated

according to the instances of the Term referenced within this multiplicity constraint by the *+gid* property. Thus, for each instance h_i of this abstract term a new relationship instance r_i is created with the *-term* set to h_i. The elements defined in the *+target* property of r are instantiated generating a set of t_j instances. Then, the elements defined in the *+source* property of r are instantiated generating a set of s_k instances. Finally, t_j is added to the *+target* property and s_k to the *+source* property of a relationship instance r_i only if the *+intersectionWithTerm* method execution result, is true. This method verifies if a given a source instance s_k or target instance t_j contain an instantiated term, i.e., an abstract term that becomes a concrete, which is equal to the term h_i of r_i or that has *originated* from h_i. Constant elements (i.e., elements that do not have abstract terms) are added to the *+target/+source* property list of all relationship instances r_i.

6.9 SUMMARY

This chapter presented a instantiation algorithm written in Java and EOL (KOLOVOS; ROSE, et al., 2013) within the Eclipse framework (ECLIPSE, 2018a) to support the automatic instantiation of executable assurance case patterns. The interfaces and abstractions proposed among modules increase the reuse of common functions and provide the basis for implementing an instantiation procedure for different SACM versions. The algorithm comprises seven instantiation modules for transforming SACM argument pattern elements in SACM product safety case elements. Therefore, executable assurance case patterns can be instantiated with traceability from abstract terms to ODE-compliant FTA elements. The next chapter evaluates the pattern extensions, methodology, tool support, and automatic instantiation of SACM executable assurance case patterns.

7 EVALUATION

This chapter describes a case study to demonstrate the feasibility of the proposed model-driven methodology in supporting the instantiation of SACM assurance case patterns from a diverse set of models (e.g., design, hazard analysis, fault trees) that constitute the Digital Dependability Identity (DDI) of an open and adaptive safety-critical system or component.

7.1 STUDY DEFINITION

The study goal was defined using the Goal-Question-Metric (GQM) goal definition template proposed by Basili et al. (BASILI; CALDIERA; ROMBACH, 1994). This study aims to analyze the **proposed model-driven assurance case methodology** for the purpose of **evaluating** it with respect to its **effectiveness and efficiency** from the point of view of the **Safety Engineers** in the context of **safety-critical systems**. The research questions and metrics shown in Table 7.1 were derived based on this goal. This study's first goal (G1) is to enhance the efficiency of the automatic instantiation of assurance case patterns. To achieve this goal, it is necessary to answer the first question (Q1) which assesses the time for generating a product assurance case. Q1 is measured by the time taken to execute the instantiation (M1) to verify the feasibility of the algorithm. The second goal (G2) focuses on ensuring the effectiveness of the automatic instantiation of assurance case patterns. To achieve this goal, it is necessary to answer the second question (Q2) which assesses the correctness of the generated assurance case. Q2 is ensured by the number of elements correctly instantiated (M2) and the number of omissions (M3). These goals aim to evaluate the capabilities of the proposed model-driven methodology in supporting the instantiation of SACM assurance case pattern concerning its correctness concerning a template solution, and execution time.

Table 7.1 – Evaluation Goals, Questions, and Metrics.

G1: Enhance the efficiency of the automatic instantiation of assurance case patterns.
Q1: How much time is needed to instantiate an assurance case pattern?
M1: Time taken to execute the instantiation.
G2: Ensuring the effectiveness of the automatic instantiation of assurance case patterns.
Q2: How correct is the instantiated product assurance case?
M2: Number of elements correctly instantiated.
M3: Number of omissions.

7.2 CASE STUDY SELECTION AND DESCRIPTION

The systems used to evaluate the proposed methodology's effectiveness and efficiency are Cypher-Physical Systems from real-world scenarios. The first system is the Hybrid Braking System (R. DE CASTRO; FREITAS, 2011) which has eleven components, sixteen ports, and twelve connections. The second system is the Highly Automated Driving Vehicle (MUNK; NORDMANN, 2020) which has seven components, twelve ports, and six connections. Both of these systems are safety-critical in the automotive domain.

The assurance case patterns used in the evaluation are the Hazard Avoidance (KELLY; MCDERMID, 1997), Risk Argument, and Hazardous Software Failure Mode (HSFM) (WEAVER, 2003). These patterns have been specified in SACM notation within the SACM ACEditor to be automatically instantiated. They have been chosen due to their distinct argument structure and reasoning approach which are crucial for ensuring software safety and reliability.

7.2.1 Hybrid Braking System

Hybrid Braking System (HBS) is a hybrid brake-by-wire system (Figure 7.1) for electric vehicles propelled by four in-wheel motors (IWMs) taken from (R. DE CASTRO; FREITAS, 2011). Hybrid means that braking is achieved through combined action of electrical IWMs, and frictional Electromechanical Brakes (EMBs). While braking, IWMs transform the vehicle's kinetic energy into electricity, which charges the power train battery, increasing the vehicle's range.

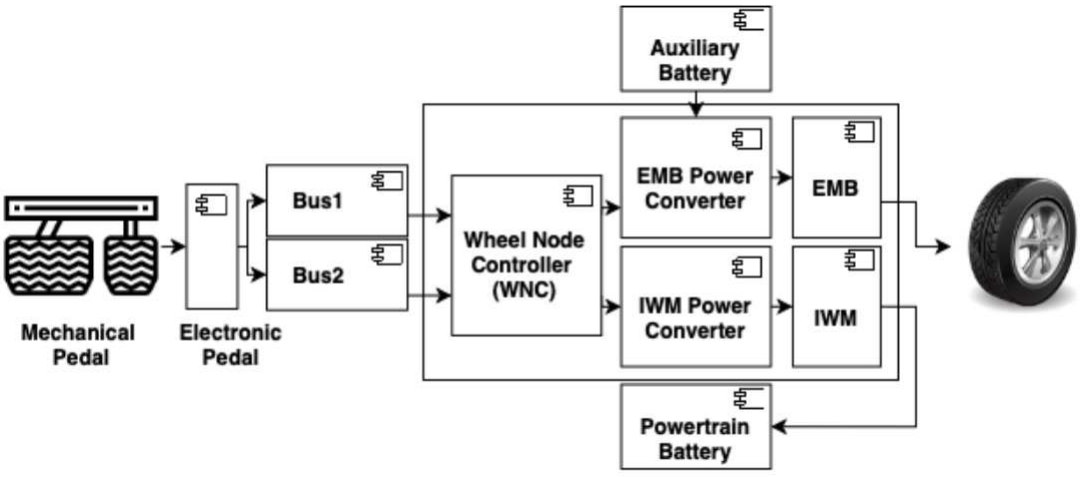

Figure 7.1 – Hybrid Braking System Architecture.

Brake-by-wire eliminates the hydraulic connection between the brake pedal and individual wheel brake modules. A redundant electronic Bus software handles the communication between the Electronic Pedal, which senses brake pedal movement, and Wheel Node Controllers (WNCs) from local Wheel Brake modules, transforming brake pedal

movement into braking torque (force) for each wheel. Auxiliary Battery provides power to each wheel brake module while braking. Power Train Battery stores the energy produced by the IWMs. The system is activated when the driver presses the mechanical pedal. The Electronic Pedal component senses the driver's action, and it sends the braking forces, via a duplex bus system, to WNCs of each wheel brake module. Each WNC generates commands to the power converters to activate EMB and IWM braking actuators. While braking, the power flows from the auxiliary battery to EMB, and from IWM to the powertrain battery. Different hazards with different criticality (i.e., ISO 26262 Automotive Safety Integrity Level - ASILs), and causes can arise from the interaction between wheel braking system components.

7.2.2 Highly Automated Driving Vehicle

Highly Automated Driving (HAD) vehicle is an automated-driven system (Figure 7.2) for electric vehicles propelled by power-train batteries (Powertrain) taken from (MUNK; NORDMANN, 2020). The HAD can sense its environment through side cameras to operate, without human involvement, the vehicle's lateral and longitudinal movement. The left and right Cameras sense the environment sending data to the VehicleComputer responsible for defining the vehicle's torque and wheel angle. The torque is sent to the Powertrain module which controls the longitudinal movement. The wheel angle is sent to the Steering module which controls the lateral movement.

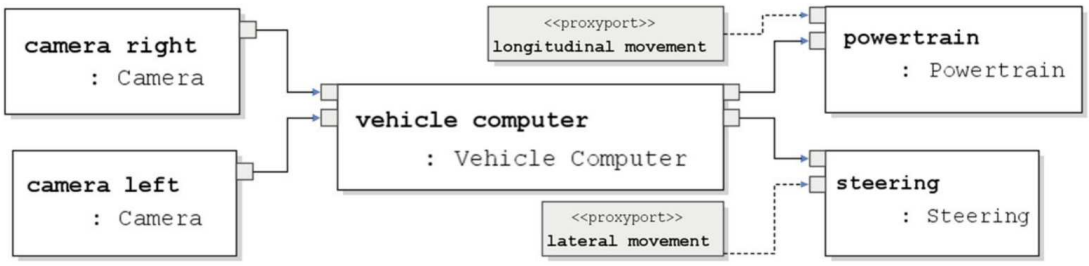

Figure 7.2 – Highly Automated Driving Vehicle Architecture.

The system is activated when the VehicleComputer identifies the necessity of changing the vehicle movement based on the images captured through the left and right Cameras. Then, the system sends the torque and angle via ports to the Powertrain and Steering modules, respectively. Thus, different hazards with different criticality (i.e., ISO 26262 Automotive Safety Integrity Level - ASILs), and causes can arise from the interaction between HAD components.

7.3 CASE STUDY EXECUTION

In this Section, each phase of the proposed methodology is performed. Section **7.3.1** shows the Assurance Case Pattern Specification phase where the assurance case

patterns were specified in SACM using the proposed patterns extension. Section **7.3.2** shows the Assurance Case Pattern Instantiation phase for two systems of the automotive domain.

7.3.1 Assurance Case Pattern Specification

This section presents the patterns which have been used in the case studies. Section *7.3.1.1* explores the Hazard Avoidance pattern, Section *7.3.1.2* the Risk Argument pattern, and Section *7.3.1.3* the Absence of Hazardous Software Failure Mode pattern.

7.3.1.1 Hazard Avoidance Pattern

The hazard avoidance pattern (KELLY; MCDERMID, 1997) decomposes the argument that the system is acceptable safe (*SysSafe*) into sub-claims arguing over the risk posed by each system hazards (*RiskHzdX*) is acceptable (Figure 7.3). This argumentation strategy (*ArgOverRiskHzds*) is in the context of the identified system hazards(*IdentHzds*). The top-level claim (*SysSafe*) is in the context of the system properties as definition (*SystemDef*), environment (*Environment*), and the target safety standard (*SafetyStandard*) to be acceptable or not acceptable as safe.

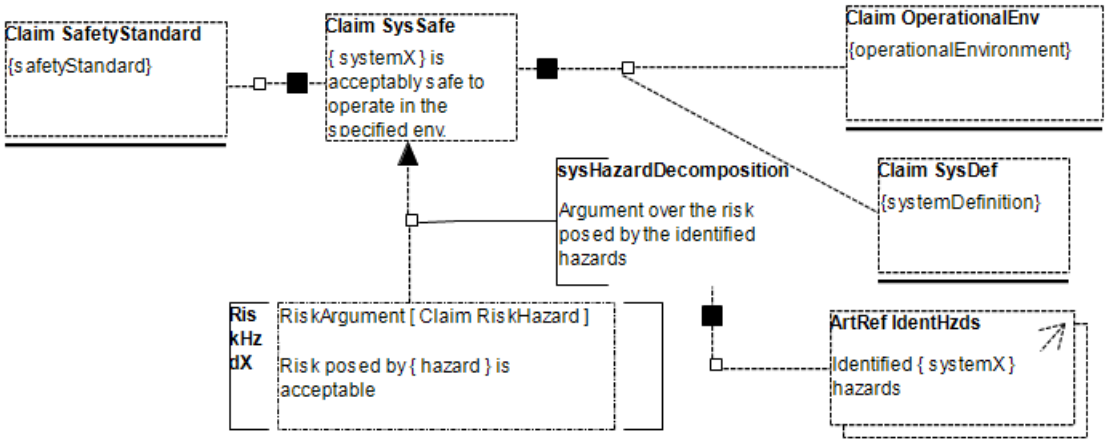

Figure 7.3 – Hazard Avoidance Pattern in SACM

Table 7.2 shows the implementation constraint subtypes assigned to the pattern abstract terms. Table 7.3 shows the queries assigned to *Mapping* implementation constraints. The term *systemX* has a *Mapping* constraint with the query *q1* which extracts from the "systems.xml" file the attribute "name" of the tag "system". The term *operationalEnvironment* has origin in *systemX* and a *Mapping* constraint with the query *q2* which extracts from the "system.xml" file the attribute "environment" of the tag "system" related to the origin. The term *systemDefinition* has origin in *systemX* and a *Mapping* constraint with the query *q3* which extracts from the "system.xml" file the attribute "sysDef" of the tag "system" related to the origin. The term *safetyStandard* has origin in *systemX* and

a *Mapping* constraint with the query *q4* which extracts from the "system.xml" file the attribute "name" of the tag "standard" related to the origin.

Table 7.2 – Hazard Avoidance Pattern Constraints.

Type	Name	Constraints	Origin	External Reference
Term	systemX	$Mapping^{q1}$		system.xml
Term	operationalEnvironment	$Mapping^{q2}$	systemX	system.xml
Term	systemDefinition	$Mapping^{q3}$	systemX	system.xml
Term	safetyStandard	$Mapping^{q4}$	systemX	system.xml

Table 7.3 – Hazard Avoidance Pattern Queries.

Query	EOL Code
q1	ExternalReference!t_system.all.collect(s|s.a_name)
q2	ExternalReference!t_system.all.selectOne(s|s.a_name="$originValue") .a_environment
q3	ExternalReference!t_system.all.selectOne(s|s.a_name="$originValue") .a_sysDef
q4	ExternalReference!t_system.all.selectOne(s|s.a_name="$originValue") .children.select(c|c.name="standard").collect(s|s.a_name)

7.3.1.2 Risk Argument Pattern

This pattern argument is over the absence of component failures that can cause a given hazard. This argument is in the context of the ASIL allocated to each system hazard stated in *SafetyStandard* of the hazard avoidance (Figure 7.4).

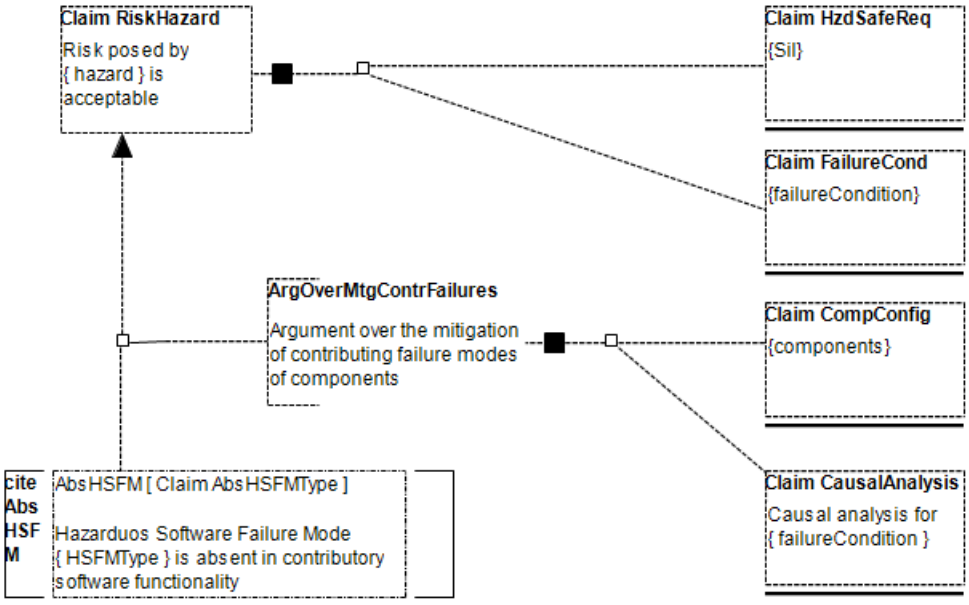

Figure 7.4 – Risk Argument pattern in SACM

The *RiskHazardX* top-level claim is stated in the context of the risk classification allocated to the system hazard in *Acceptable*, and the top-level failure condition leading to this hazard in claim *FailureCondition*. The top-level claim is decomposed into sub-claims arguing the mitigation of component failures that directly contribute to the occurrence of this hazard. Such decomposition strategy is defined in the context of the causal chain defined in the hazard fault tree. The elements of this chain are decomposed by *AbsHSFM*, which is supported by sub-claims arguing the absence of each contributing hazardous software failure mode.

Table 7.4 – Risk Argument Pattern Constraints.

Type	**Name**	**Constraints**	**Origin**	**External Reference**
Term	hazard	$Mapping^{q1}$, *Multiplicity*	systemX	system.xml
Term	Sil_1	$Mapping^{q2}$, *Multiplicity*	hazard	system.xml
Term	failureCondition	$Mapping^{q3}$, *Multiplicity*	hazard	system.xml
Term	components	$Mapping^{q4}$, *Multiplicity*	hazard	system.xml

Table 7.4 shows the implementation constraint subtypes assigned to the pattern abstract terms. Table 7.5 shows the queries assigned to *Mapping* implementation constraints. The term *hazard* has origin in *systemX*, a *Multiplicity* constraint, and a *Mapping* constraint with the query *q1*. The query *q1* extracts from the "systems.xml" file the attribute "name" of "hazard" tags within the "system" tag related to the origin. The term *Sil* has its origin in *hazard*, a *Multiplicity* constraint, and a *Mapping* constraint with the query *q2*. The query *q2* extracts from the "system.xml" file the attribute "sil" of the tag "hazard" related to the origin. The term *failureCondition* has its origin in *hazard*, a *Multiplicity* constraint, and a *Mapping* constraint with the query *q3*. The query *q3* extracts from the "system.xml" file the attribute "text" of "cause" tags within the "hazard" tag related to the origin. The term *components* has its origin in *hazard*, a *Multiplicity* constraint, and a *Mapping* constraint with the query *q4*. The query *q4* extracts from the "system.xml" file the attribute "text" of "component" tags within the "hazard" tag related to the origin.

Table 7.5 – Risk Argument Pattern Queries.

Query	**EOL Code**
q1	ExternalReference!t_system.all.selectOne(s|s.a_name="$originValue") .children.select(c|c.name="hazard").collect(s|s.a_name)
q2	ExternalReference!t_hazard.all.selectOne(h|h.a_name="$originValue").a_sil
q3	ExternalReference!t_hazard.all.selectOne(h|h.a_name="$originValue") .children.select(c|c.name="cause").collect(t|t.text)
q4	ExternalReference!t_hazard.all.selectOne(h|h.a_name="$originValue") .children.select(c|c.name="component").collect(t|t.text)

7.3.1.3 HSFM Pattern

An Absence Hazardous Software Failure Mode (HSFM) fault mitigation pattern argument is over the occurrence of primary, secondary, and control failure modes of a given fault tree gate (e.g., AND/OR gates) do not lead the system to an unsafe state (Figure 7.5). This pattern decomposes the claim *ABSHSFMType* into tree sub-claims: *i) AbValPrimary* arguing that the current failure mode is acceptable; *ii) AbsTypePrimarySecondary* arguing that the failure modes of other components that contribute to the current failure mode are acceptable; *iii) AbsTypeControl* arguing that the contributory software functionality component is scheduled and allowed to run once. The *AbsTypeSecondary* is further decomposed into fault mitigation sub-claims (*HSFMAccept*) arguing that all causes of each failure event specified in fault tree leaf nodes are acceptable, i.e., they do not lead the system to an unsafe state. For each fault tree non-leaf node, the *AbsHSFMType* is decomposed into another "Absence Hazardous Software Failure Mode" (HSFM) fault mitigation argument.

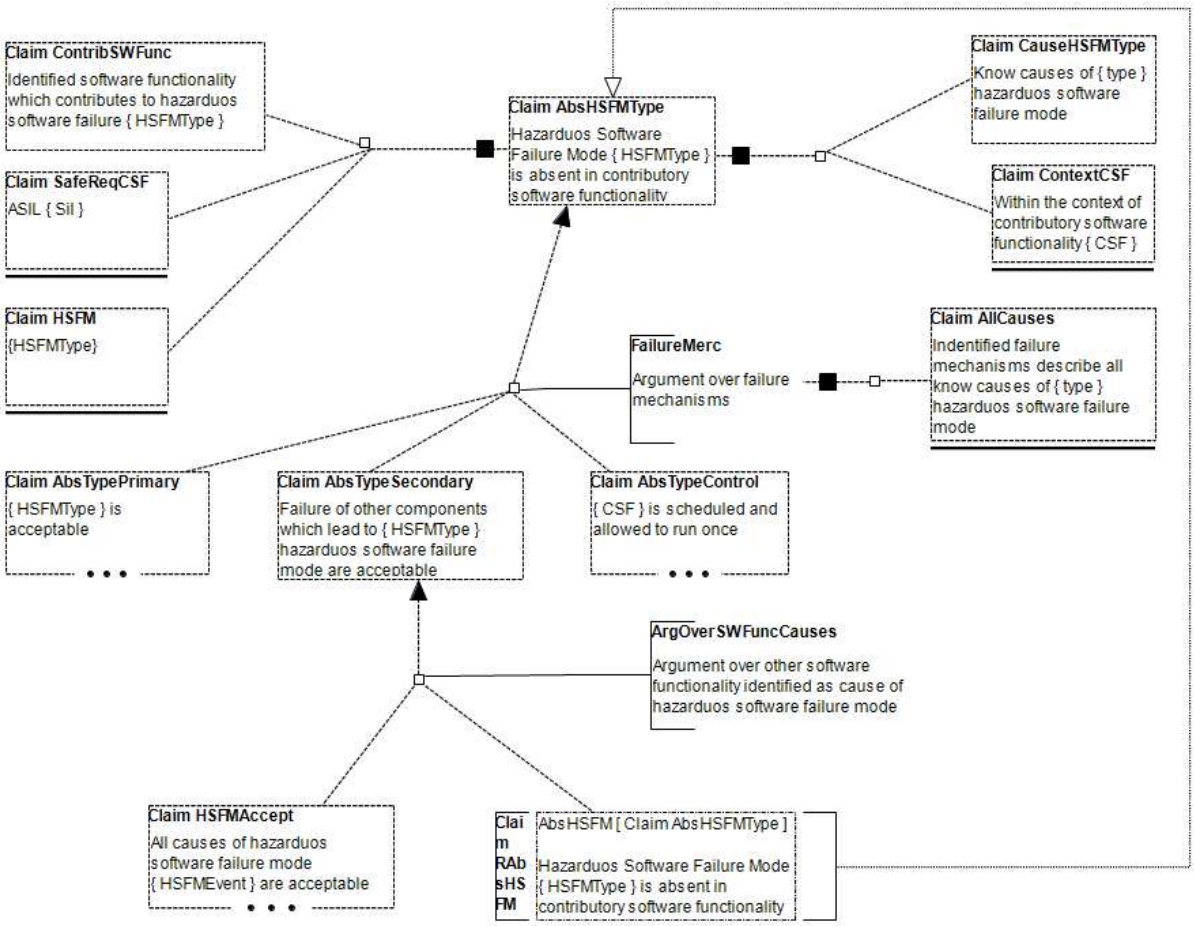

Figure 7.5 – HSFM Pattern in SACM

Table 7.6 shows the implementation constraint subtypes assigned to the pattern abstract terms. Table 7.7 shows the queries assigned to *Mapping* and *Children* implementation constraints. The term *HSFMType* has origin in *hazard*, a *Multiplicity* constraint,

a *Mapping* constraint with the query *q1*, and a *Children* constraint with the query *q2*. The query *q1* extracts from the "ODE.model" file the *+name* property of Cause elements with *+description* property containing "TOP-EVENT" within the FaultTree related to the origin. The query *q2* extracts from the "ODE.model" file the *+name* property of Gate elements within the *+causes* property of the Gate related to the parent. The term *Sil* has its origin in *HSFMType*, a *Multiplicity* constraint, and a *Mapping* constraint with the query *q3*. The query *q3* extracts from the "ODE.model" the property *+value* of a KeyValueMap with the *+key* set to "sil" within the *+keyValueMaps* property of the Cause related to the origin.

Table 7.6 – HSFM Pattern Constraints.

Type	**Name**	**Constraints**	**Origin**	**External Reference**
Term	HSFMType	$Mapping^{q1}$, $Children^{q2}$, $Multiplicity$	hazard	ODE.model
Term	Sil_2	$Mapping^{q3}$, $Multiplicity$	HSFMType	ODE.model
Term	type	$Mapping^{q4}$, $Multiplicity$	HSFMType	ODE.model
Term	CSF	$Mapping^{q5}$, $Multiplicity$	HSFMType	ODE.model
Term	HSFMEvent	$Mapping^{q6}$, $Multiplicity$	HSFMType	ODE.model

Table 7.7 – HSFM Pattern Queries.

Query	**EOL Code**
q1	ExternalReference!FaultTree.all.selectOne(s|s.name="$originValue") .causes.select(c|c.description.contains("TOP-EVENT")) .collect(s|s.name)
q2	ExternalReference!Gate.all.selectOne(s|s.name="$parentValue")<>null ? ExternalReference!Gate.all.selectOne(s\|s.name="$parentValue") .causes.select(a|a.causeType.name = "Gate").collect(c|c.name) : Sequence{}
q3	Sequence{ExternalReference!Cause.all.selectOne(s|s.name="$originValue") .keyValueMaps.selectOne(s|s.key="sil")?.values? .selectOne(v|v.tag="Safety Requirement")?.value}
q4	ExternalReference!Cause.all.selectOne(s|s.name="$originValue") .failure.collect(s|s.failureClass)
q5	Sequence{"$originValue".replace("\.[A-z]*[0-9]*$|[A-z]*-","")}
q6	ExternalReference!Gate.all.selectOne(s|s.name="$parentValue")<>null ? ExternalReference!Gate.all.selectOne(s\|s.name="$parentValue") .causes.select(a|a.causeType.name <> "Gate").collect(c|c.name) : Sequence{}

The term *type* has its origin in *HSFMType*, a *Multiplicity* constraint, and a *Mapping* constraint with the query *q4*. The query *q4* extracts from the "ODE.model" file the *+failureClass* property of the Failure associated with the Cause related to the origin. The term *CSF* has its origin in *HSFMType*, a *Multiplicity* constraint, and a *Mapping* constraint

with the query *q5*. The query *q5* returns the origin value without separators. The term *HSFMEvent* has its origin in *HSFMType*, a *Multiplicity* constraint, and a *Mapping* constraint with the query *q6*. The query *q6* extracts from the "ODE.model" file the *+name* property of Cause elements with *+causeType* different from "Gate" within the *+causes* property of the Gate related to the origin.

7.3.2 Assurance Case Pattern Instantiation

This section describes the application of the Assurance Case Pattern Instantiation phase in the target systems. Section *7.3.2.1* presents the instantiation of the specified patterns for the Hybrid Braking System (HBS). Section *7.3.2.2* presents the instantiation of the specified patterns for the Highly Automated Driving Vehicle (HAD).

7.3.2.1 HBS Safety Analysis and Pattern Instantiation

The system safety analysis was executed resulting in the identification of three hazards: No braking after a request from the driver (**H1: No Braking Four Wheels, ASIL D**); No front wheels braking after request (**H2: No Braking Front, ASIL C**); and No rear wheels braking after request (**H3: No Braking Rear, ASIL B**). Figure 7.6 shows a fault tree except for the hazard **No Braking Four Wheels**. At the first level is the top event *No_Breaking_4_Wheels*. At the second level, there are eight intermediate events related to the EMB.Out1 and IWM.Out1 of each break unit. Finally, at the third level, there is one intermediate event *Omission-Brake_Unit4.Wheel_Node_Controller.Out2* and three basic events *Brake_Unit4.IWM.OFailure*, *Powertrain_Battery.OFailure*, and *Brake_Unit4.IWM_Power_Converter.OFailure*.

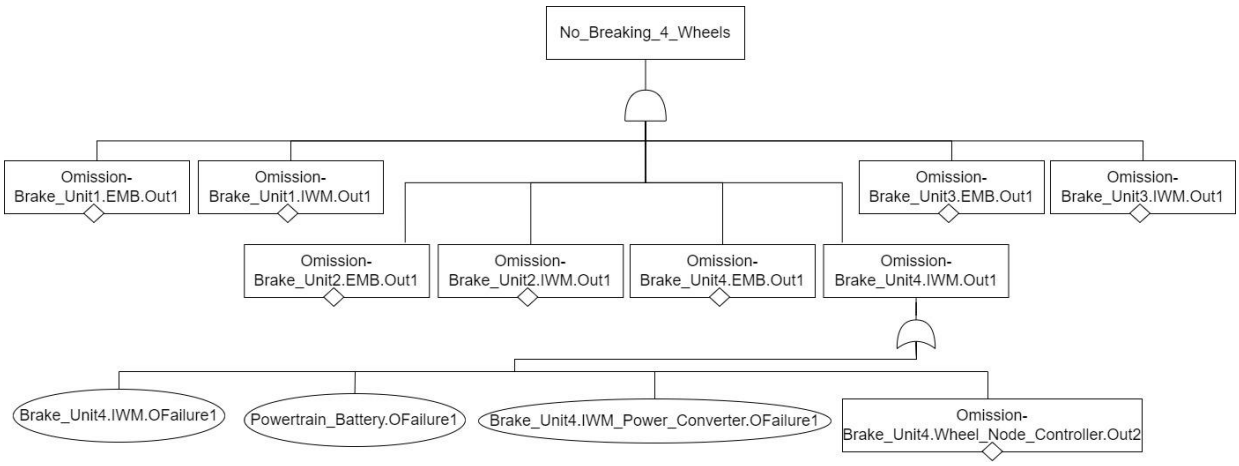

Figure 7.6 – HBS Fault Tree Excerpt.

After the execution of the safety analysis, the fault tree model specified using HiP-HOPs is converted into the Open Dependability Exchange (ODE) metamodel-compliant (Figure 7.7). In this transformation process, the top and intermediate events of the fault

tree are converted into ODE Gate elements with the *+causes* property set to their children. The fault tree basic events are represented through the ODE Cause elements. ODE Failure elements are created for the fault tree nodes with details about their failure type (i.e., *+failureClass* property).

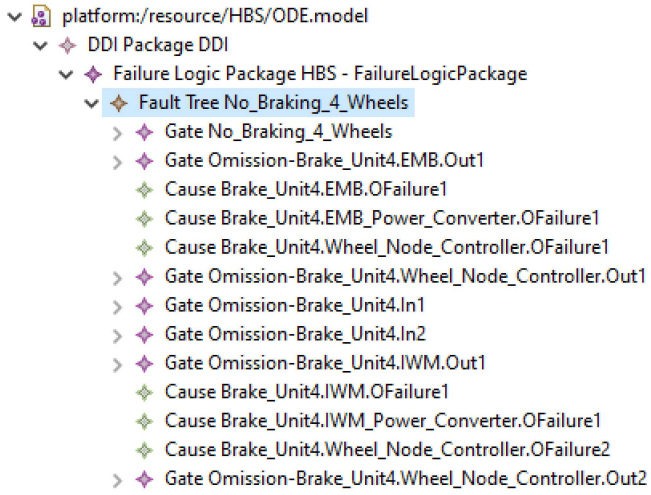

Figure 7.7 – HBS ODE Fault Tree Excerpt.

The HBS architecture and identified hazards have been represented in an XML file (Listing 7.1). This file has a "system" tag with "subsystem" and "hazard" tags to describe some architectural and analysis information. Thus, this file is used to represent the general information regarding the system and hazards, i.e., definition, environment name, sil, components, and causes.

Listing 7.1 – HBS System Model Exerpt.

```
<system name="HBS" sysDef="Definition of HBS" enviroment="high speed roads">
        <standard name="ISO 26262"/>
        <subsystem name="MechanicalPedal">
                <Port type="out" name="Out1"/>
        </subsystem>
        <hazard name="No_Braking_4_Wheels" sil="4"
                within="HazardAnalisys" faulttree="FT_No_Braking_4_Wheels">
                <cause>
                        Omission-Brake_Unit1.Add.Braking AND
                        Omission-Brake_Unit2.Add.Braking AND
                        Omission-Brake_Unit3.Add.Braking AND
                        Omission-Brake_Unit4.Add.Braking
                </cause>
                <component>Brake_Unit1</component>
                <component>Brake_Unit2</component>
                <component>Brake_Unit3</component>
                <component>Brake_Unit4</component>
        </hazard>
```

The automatic instantiation result is a product assurance case comprising HazardAvoidance, Risk Argument, and HSFM patterns. The Hazard avoidance (Figure 7.8) is instantiated based on information within the HBS "system.xml" file described above.

The name of the system has been instantiated correctly within the claims *SysSafe* and *IdentHzds*. The context of the argumentation has been instantiated according to the HSB context given by the definition of HBS (*SysDef*), high-speed roads operational environment (*operationalEnvironment*), and the target safety standard ISO 26262 (*SafetyStandard*). The citation claim *RiskHzdX* has been instantiated for *No_Breaking_Front*, *No_Breaking_Rear*, and *No_Breaking_4_Wheels* identified HBS hazards. These citation claims cite claims within argument packages instantiated based on the *Risk Argument* pattern.

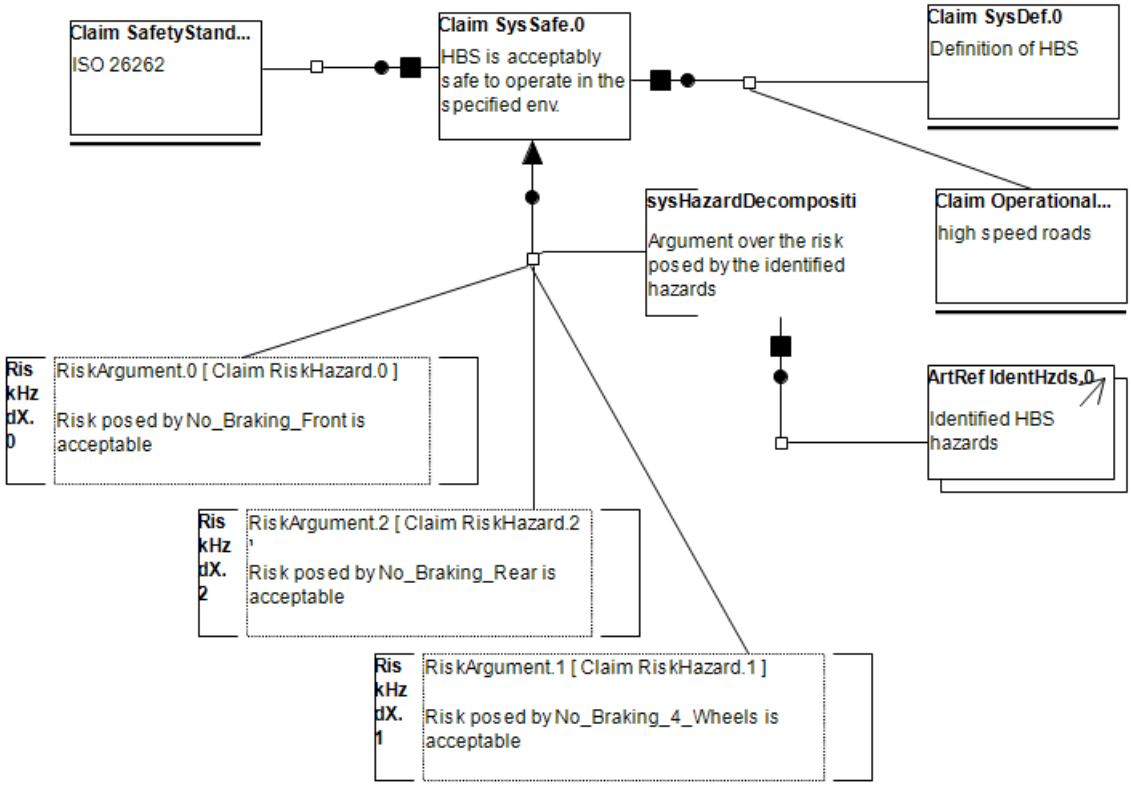

Figure 7.8 – HBS Hazard Avoidance.

The Risk Argument is also instantiated based on information within the HBS "system.xml" file. This pattern is instantiated for each identified HBS hazard describing the acceptance of the risk posed by them. Figure 7.9 shows the instantiation result for the hazard **H1: No Braking Four Wheels, ASIL D**. The name of the hazard has been instantiated correctly within the claim *RiskHazard*. This claim is in the context of the numeric value of safety integrity level D (*HzdSafeReq*), and the condition of omission of all break unit modules in add braking (*FailureCond*). The reasoning for the argument decomposition is in the context of the causal analysis of the failure condition (*CausalAnalysis*) and the brake units involved in this failure (*CompConfig*). The citation claim *citeAbsHSFM* has been instantiated for the *No_Breaking_4_Wheels* fault tree top event. This citation claim cites a claim within an argument package (*AbsHSFM.17)* instantiated based on the HSFM pattern.

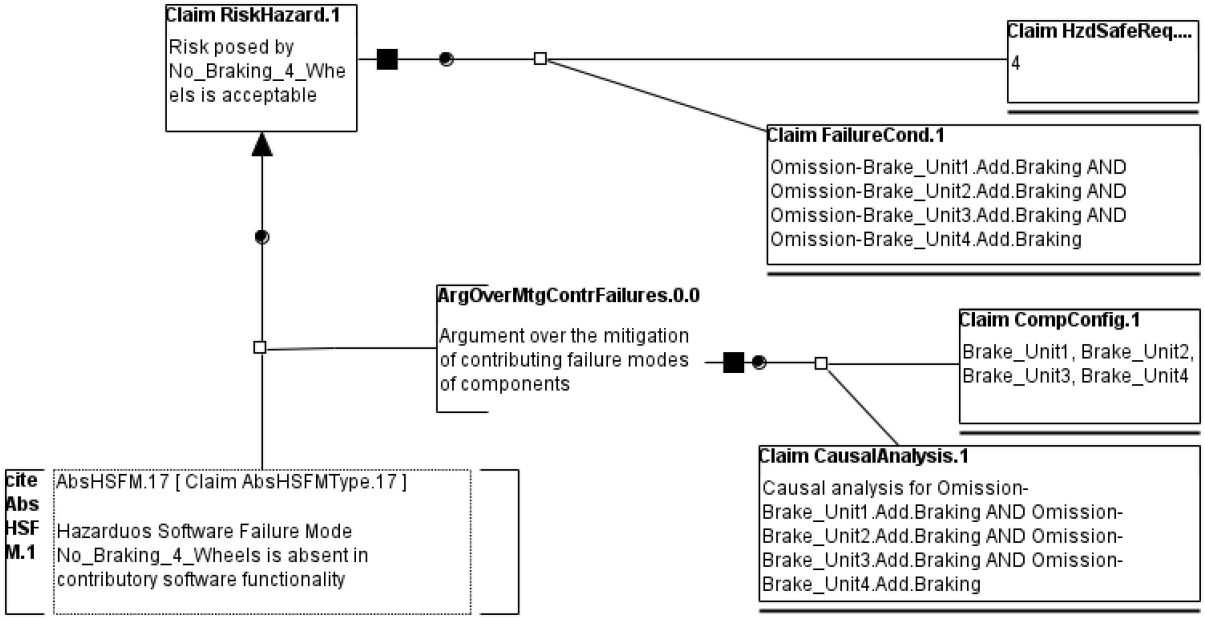

Figure 7.9 – HBS Risk Argument.

The HSFM pattern is instantiated based on HBS ODE fault tree model file information. This pattern is instantiated for each top and intermediate event of the fault tree. Figure 7.10 shows the instantiation result for the top-event "No_Breaking_4_Wheels". The *No_Breaking_4_Wheels* failure mode does not lead the system to an unsafe state (*AbsHSFMType*). This claim is in the context of the safety integrity level D (*SafeReqCSF*), identified and contributory software functionalities (*ContribSWFunc*, *ContextCSF*), hazardous software failure mode (*HSFM*), and the failure type (i.e., *+failureClass*) associated with the failure mode (*CauseHSFMType*, *AllCauses*). The argumentation is decomposed into citation sub-claims comprising the failures that contribute to the *No_Breaking_4_Wheels* failure. In this case, there are eight different citation claims related to failures in the EMB.Out1 and IWM.Out1 ports of each break unit. They represent intermediate events of the fault tree, thus, they cite claims within argument packages instantiated recursively based on the HSFM pattern.

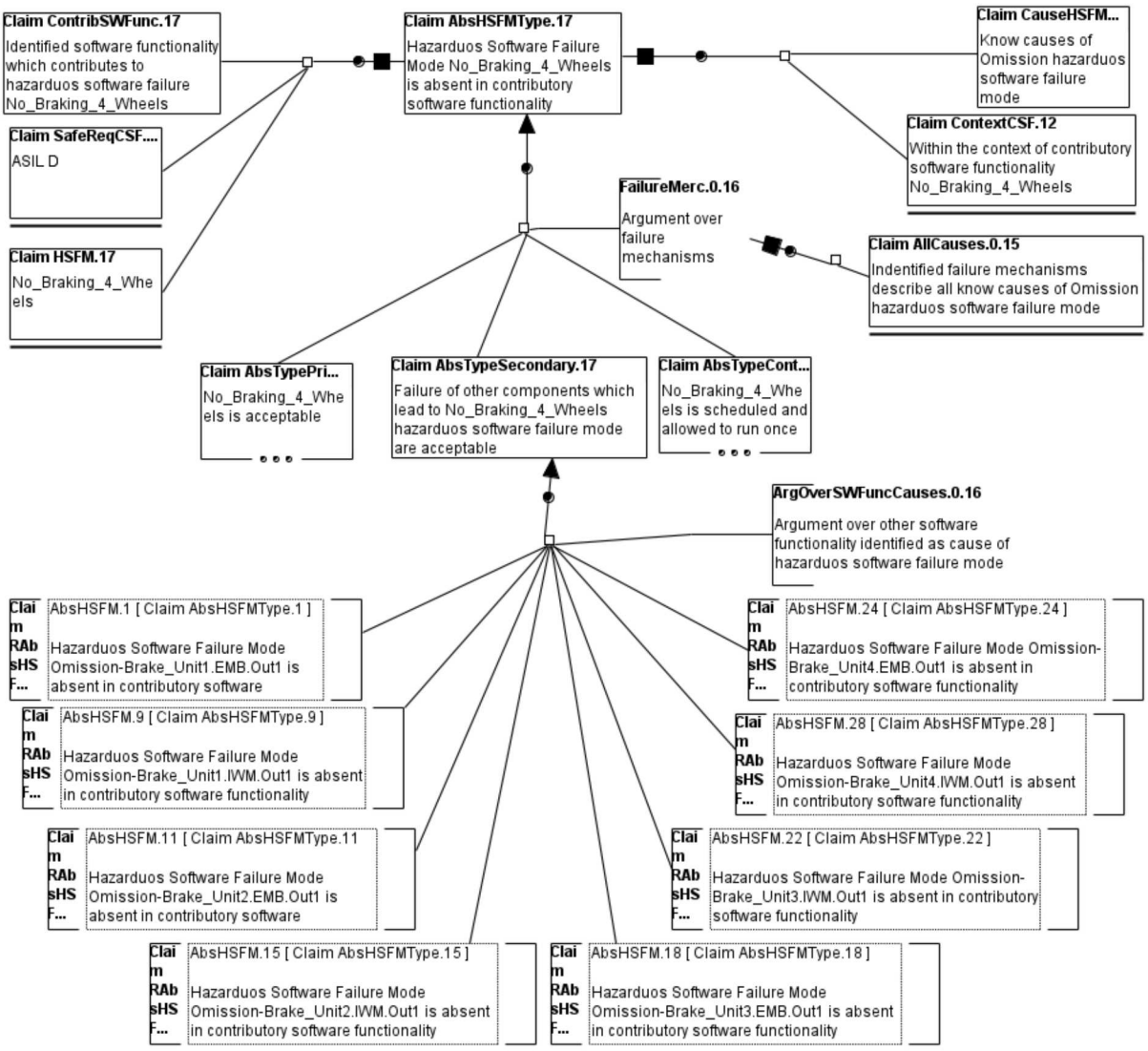

Figure 7.10 – HBS HSFM.

7.3.2.2 HAD Safety Analysis and Pattern Instantiation

The system safety analysis resulted in the identification of one hazard **Hazarduos Movement, ASIL D**. This result has been presented and described by (MUNK; NORDMANN, 2020). Figure 7.11 shows a fault tree except for the hazard **Hazarduos Moviment**. At the first level is the top event *hazardous movement* which can be caused by *wrong lateral movement* or *wrong longitudinal movement*. The paths for these failures are broken down until the basic events. The source of the *wrong lateral movement* can be an *internal fault* in the vehicle computer. The *Powetrain torque* may cause the *wrong longitudinal movement* due *too high torque* or *too low torque*. The torque failures can be caused by an *internal fault* in the vehicle computer or *blurry image*. A failure in the left and right cameras can lead to a *blurry image*.

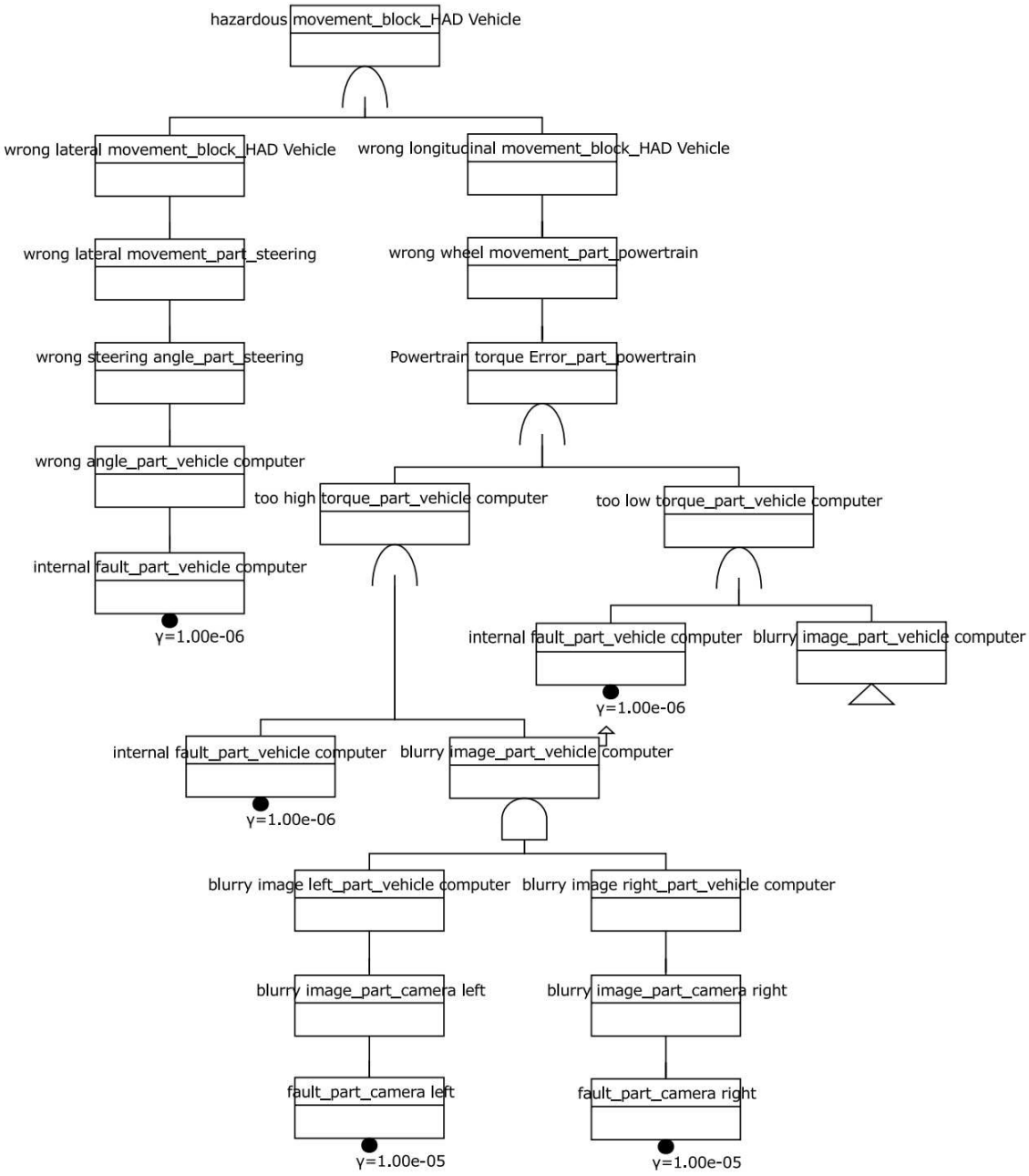

Figure 7.11 – HAD Fault Tree Excerpt (MUNK; NORDMANN, 2020).

The automatic transformation step from fault tree to Open Dependability Exchange (ODE) compliant model could not be performed due to the lack of a HiP-HOPs model. To contour this problem the fault tree model has been manually created within an ODE model (Figure 7.12). In the same line as the automatic transformation, the top and intermediate events of the fault tree are created in the form of ODE Gate elements with the *+causes* property set to their children. The fault tree basic events are created in the form of ODE Cause elements. ODE Failure elements are specified for the fault tree nodes with details about their failure type (i.e., *+failureClass* property).

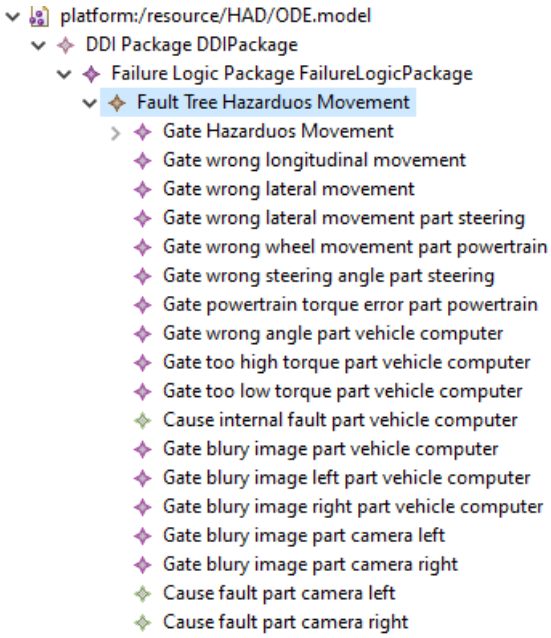

Figure 7.12 – HAD ODE Fault Tree Excerpt.

The HAD architecture and identified hazards have been represented in an XML file (Listing 7.2). This file has a "system" tag with "subsystem" and "hazard" tags to describe some architectural and analysis information. Thus, this file is used to represent the general information regarding the system and hazards, i.e., definition, environment name, sil, components, and causes.

Listing 7.2 – HAD System Model Exerpt.

```
<system name="HAD" sysDef="Definition of HAD" enviroment="high speed roads">
        <standard name="ISO 26262"/>
        <subsystem name="longitudinal movement">
                <Port type="out" name="Out1"/>
        </subsystem>
        <subsystem name="lateral movement">
                <Port type="out" name="Out1"/>
        </subsystem>
        <hazard name="Hazarduos Movement" sil="4"
 within="HazardAnalisys" faulttree="FT_hazarduos movement">
                <cause>
                        wrong longitudinal movement AND
                        wrong lateral movement
                </cause>
                <component>longitudinal movement</component>
                <component>lateral movement</component>
        </hazard>
```

The automatic instantiation result is a product assurance case comprising HazardAvoidance, Risk Argument, and HSFM patterns. The Hazard avoidance (Figure 7.13) is instantiated based on information within the HAD "system.xml" file described above. The name of the system has been instantiated correctly within the claims *SysSafe* and *IdentHzds*. The context of the argumentation has been instantiated according to the HAD

context given by the definition of HAD (*SysDef*), high-speed roads operational environment (*operationalEnvironment*), and the target safety standard ISO 26262 (*SafetyStandard*). The citation claim *RiskHzdX* has been instantiated for *Hazardous Movement* identified hazard. This citation claim cites a claim within an argument package instantiated based on the *Risk Argument* pattern.

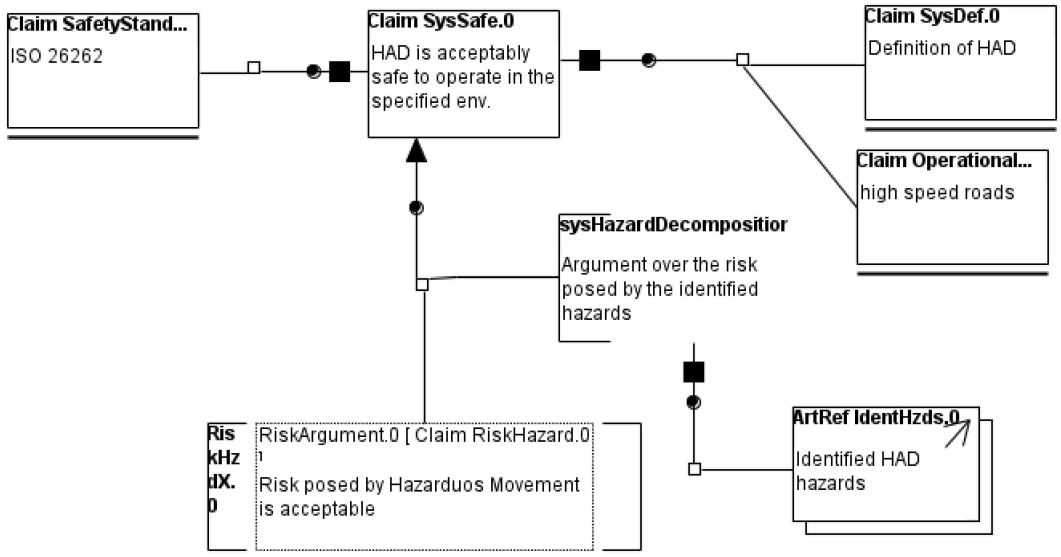

Figure 7.13 – HAD Hazard Avoidance.

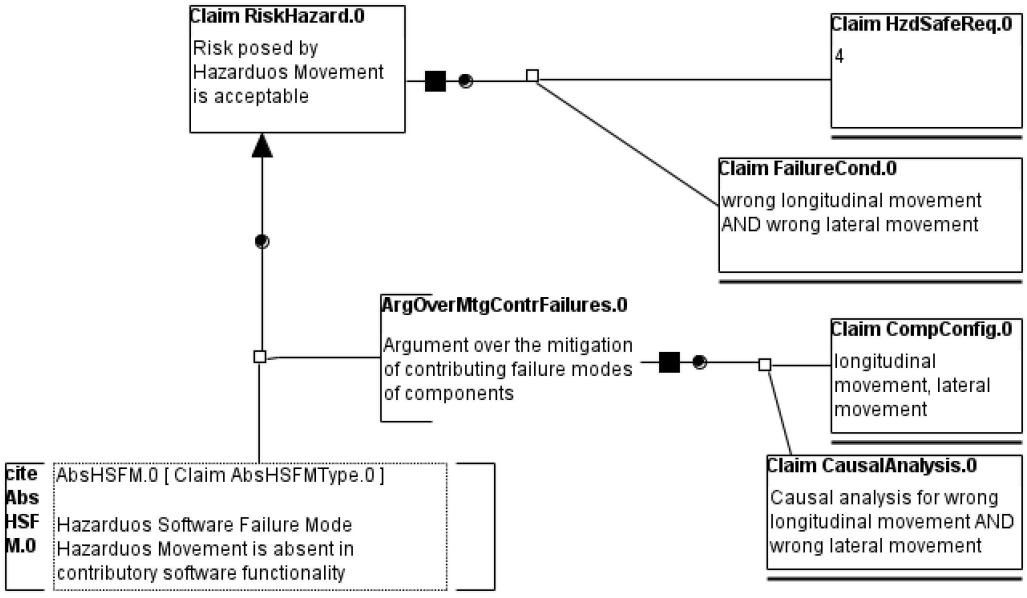

Figure 7.14 – HAD Risk Argument.

The Risk Argument is also instantiated based on information within the HAD "system.xml" file. This pattern is instantiated for each identified HAD hazard describing the acceptance of the risk posed by them. Figure 7.14 shows the instantiation result for the hazard **H1: Hazardous Movement, ASIL D**. The name of the hazard has been

instantiated correctly within the claim *RiskHazard*. This claim is in the context of the numeric value of safety integrity level D (*HzdSafeReq*), and the condition of wrong lateral and longitudinal movement (*FailureCond*). The reasoning for the argument decomposition is in the context of the causal analysis of the failure condition (*CausalAnalysis*), the lateral, and longitudinal components involved in this failure (*CompConfig*). The citation claim *citeAbsHSFM* has been instantiated for the*Hazardous Movement* fault tree top event. This citation claim cites a claim within an argument package (*AbsHSFM.0)* instantiated based on the HSFM pattern. The HSFM pattern is instantiated based on information within the HAD ODE fault tree model file. This pattern is instantiated for each top and intermediate event of the fault tree. Figure 7.15 shows the instantiation result for the top-event Hazardous Movement".

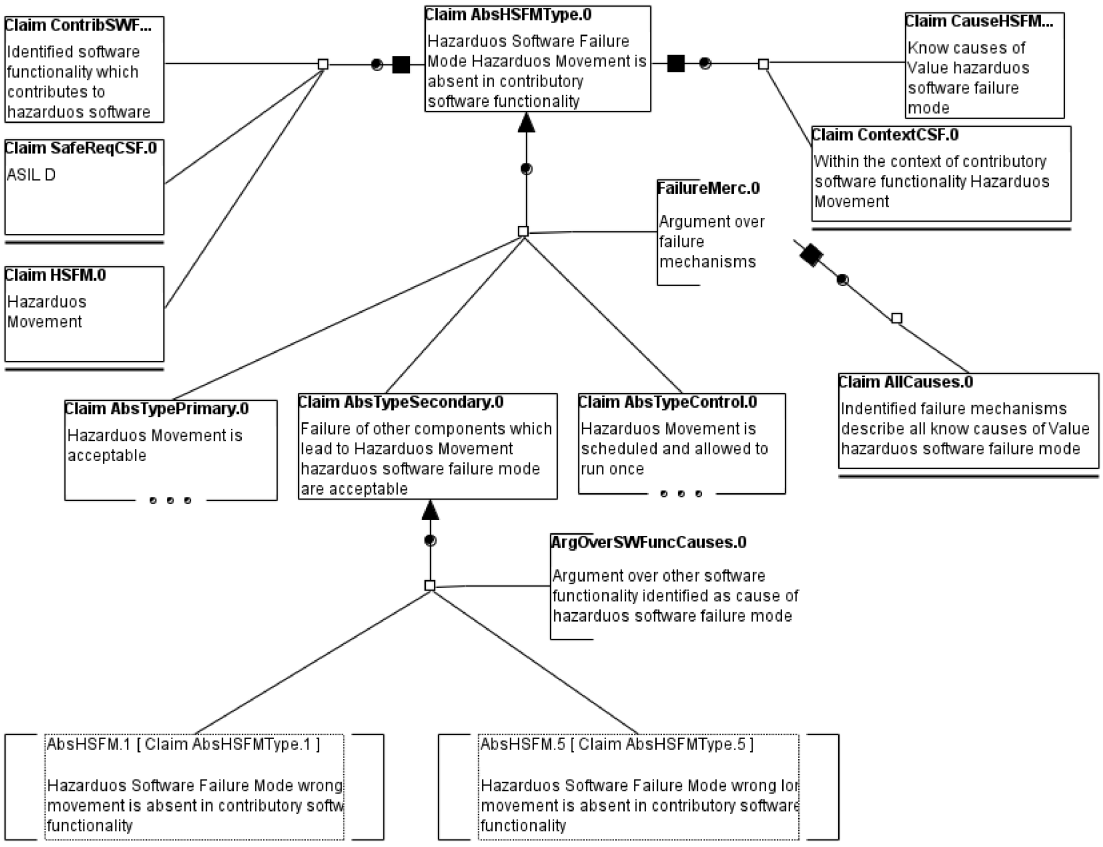

Figure 7.15 – HAD HSFM.

The *Hazardous Movement* failure mode does not lead the system to an unsafe state (*AbsHSFMType*). This claim is in the context of the safety integrity level D (*SafeReqCSF*), identified and contributory software functionalities (*ContribSWFunc*, *ContextCSF*), hazardous software failure mode (*HSFM*), and the failure type (i.e., *+failureClass*) associated with the failure mode (*CauseHSFMType*, *AllCauses*). The argumentation is decomposed into citation sub-claims comprising the failures that contribute to the *Hazardous Movement* failure. In this case, there are two citation claims related to failures in the lateral and

longitudinal movements. These claims represent intermediate events of the fault tree, thus, they cite claims within argument packages instantiated recursively based on the HSFM pattern.

7.4 DATA COLLECTION AND ANALYSIS

In order to **enhance the efficiency of the automatic instantiation of assurance case patterns** (**G1**) the Hazard Avoidance, Risk Argument, and HSFM assurance case patterns, have been instantiated thirty times in a single computer. This computer has a Windows operational system, processor Intel(R) Core(TM) i3-8100 3.60 GHz, and 8.00 gigabytes of RAM. The execution times were measured using the `System.currentTimeMillis()` function in Java. The data have been collected after each execution outputs the result and the time taken to instantiate the patterns. Table *7.8* shows **how much time is needed to instantiate an assurance case pattern** (**Q1**). For the HBS system, the minimum time was 1710 milliseconds, the maximum time was 2250 milliseconds, and the average time was 1759 milliseconds. For the HAD system, the minimum time was 189 milliseconds, the maximum time was 279 milliseconds, and the average time was 205.5 milliseconds. The **time taken to execute the instantiation** (**M1**) with 95% confidence falls between 1722.43 and 1795.57 milliseconds for HBS and 196.8 and 214.2 milliseconds for HAD.

Table 7.8 – Instantiation Efficiency.

System	Exec.	Max.(ms)	Min.(ms)	Avg.(ms)	M1(ms)	M1(s)
HBS	30	2250	1710	1759	1759 ± 36.57	1.76 ± 0.04
HAD	30	279	189	205.5	205.5 ± 8.7	0.21 ± 0.01

The automatic instantiation process is deterministic unless the external artifacts or the assurance case patterns change. Therefore, the result of the instantiation is the same no matter how many times the pattern is instantiated. Thus, in order to **ensuring the effectiveness of the automatic instantiation of assurance case** (**G2**), is necessary to compare the generated assurance case with the expected result. In this study, the elements of interest to be compared are Claim, ArgumentReasoning, ArtifactReference, and AssertedRelationship instances. Table 7.9 shows **how correct is the instantiated product assurance case** (**Q2**). The total **number of elements correctly instantiated** (**M2**) are 662 for the HBS system and 248 for the HAD system. The **total number of omissions** (**M3**) are 4 for the HBS system and 12 for the HAD system. For the patterns Hazard Avoidance and Risk Argument all the elements have been correctly instantiated and no omissions were detected. However, not all the elements have been correctly instantiated for the HSFM pattern. Four omissions have been detected for the HBS system and twelve for the HAD system.

Table 7.9 – Instantiation Effectiveness.

	Hazard Avoidance		**Risk Argument**		**HSFM**		**Total**	
	HBS	**HAD**	**HBS**	**HAD**	**HBS**	**HAD**	**HBS**	**HAD**
M2	13	11	30	10	619	227	662	248
M3	0	0	0	0	4	12	4	12

An analysis has been executed and the cause for these omissions has been found. The omissions detected in the HBS instantiation are due to the automatic transformation from the fault tree model to an ODE-compliant model. The *+failureClass* property of ODE Failure is only filled when the fault tree intermediate event name is prefixed with the failure type (e.g., Omission, Value). Thus, for these non-prefixed events, the generated ODE Failure associated with the respective Gate does not have a *+failureClass*. Therefore, the HSFM pattern has not been fully instantiated due to the lack of values for the abstract term *type* within claims *CauseHSFMType* and *AllCauses*. Thus, the FTA event names must be prefixed by the failure type to be correctly instantiated by the implemented transformation. The omissions detected in the HAD instantiation are due to the manually generated ODE-compliant fault tree. The safety integrity level has not been defined in the property *+value* of a KeyValueMap within ODE Failures that have been created from fault tree intermediate events.

The proposed automatic instantiation has achieved good results. The efficiency (**G1**) regarding **how much time is needed to instantiate an assurance case pattern** (**Q1**) has been addressed by the low instantiation time (**M1**). The effectiveness (**G2**) regarding **how much correct is the instantiated product assurance case** (**Q2**) has been addressed by the high number of elements correctly instantiated (**M2**) and the low number of omissions (**M3**).

7.5 THREATS TO THE VALIDITY

This section presents the identified threats to the validity for the performed case study. Construct validity (Section **7.5.1**) concerns the extent to which the measures accurately represent the concepts they are intended to measure. External validity (**7.5.2**) refers to the generalizability of the findings beyond the specific context of the study and sample size. Reliability (**7.5.3**) pertains to the consistency and reproducibility of the results.

7.5.1 Construct Validity

Measurement Errors: Inaccurate measurement of the effectiveness or efficiency of the instantiated patterns can affect the validity of the results. This threat has been

mitigated by providing **M1**, **M2**, and **M3** metrics to quantify the efficiency and effectiveness to automatically instantiate assurance case patterns.

7.5.2 External Validity

Generalizability: The findings from the case study might not be applicable to other types of systems or contexts. This thesis provided a case study with two cyber-physical from the automotive domain. However, the proposed automatic instantiation is not limited to specific systems or domains. The process depends on the specification of executable assurance case patterns and the traceability to external artifacts. Therefore, there is the possibility of generalizing the findings for the automotive domain in general and also for other domains.

Sample Size: A small sample size or a limited number of case studies can reduce the ability to generalize the findings to other assurance case patterns. In order to mitigate this threat, three real-world SACM assurance case patterns have been specified and instantiated. Thus, a large set of SACM features, pattern extension constraints, and instantiation capabilities have been explored.

7.5.3 Reliability

Reproducibility: The process of automatic instantiation must be reproducible to ensure the reliability of the findings. If the results cannot be consistently replicated, the reliability is compromised. Due to the deterministic nature of the automatic instantiation, the results can be reliably replicated. This means that the generated product assurance case remains consistent when the same pattern and external artifacts are instantiated multiple times.

7.6 SUMMARY

This chapter presented a comprehensive evaluation of the automatic instantiation of executable assurance case patterns. The study focused on two Cyber-Physical Systems: the Hybrid Braking System (HBS) and the Highly Automated Driving (HAD) system. These systems were chosen due to their complexity and relevance in the automotive domain. The assurance case patterns, specifically the Hazard Avoidance, Risk Argument, and Hazardous Software Failure Mode (HSFM) patterns, were meticulously specified using the proposed methodology. These patterns were further enhanced with the proposed SACM pattern extensions within the SACM ACEditor tool to ensure they met the necessary requirements for instantiation.

Following the specification, the SACM patterns were instantiated for both the HBS and HAD systems. This process involved applying the patterns to the systems to generate

assurance cases that demonstrate the safety and reliability of the systems. The results of these instantiations were then presented in detail for each system. This included a thorough analysis of the automatic instantiation process, highlighting the efficiency and effectiveness of the methodology. Additionally, the chapter provided a detailed description of the threats to the validity of the study, ensuring that the findings are robust and reliable. This evaluation not only demonstrates the feasibility of the automatic instantiation but also provides insights into its potential applications in other domains.

8 CONCLUSION

Assessing confidence in CPS dependability properties (e.g., safety and security) is essential to protect people, the environment, and property from unintentional harm (safety) and against intentional threats (security). Due to the open and adaptive nature of CPS, it demands a paradigm shift from design-time to runtime system assurance. As a standardized and well-structured metamodel, SACM provides the foundations for runtime assurance. This work extended SACM with **pattern extensions** to add semantics to the concept of Implementation Constraint providing support for traceability between assurance cases and system design, analysis, and process models, which are part of Executable Digital Dependability Identities (EDDIs). This concept has been used in a **assurance case editor tool** and a novel **model-driven methodology** in order to support the specification and synthesis of SACM-compliant executable assurance cases. To concertize the synthesis of assurance cases, a **instantiation algorithm** has been implemented to automatically instantiate executable assurance case patterns to demonstrate the safety and security of CPS at runtime.

The **pattern extensions** facilitate the creation of executable argument patterns linked to evidence within the SACM. This contribution proposes five sub-types of implementation constraints (i.e., multiplicity, optionality, choice, mapping, and children) that can be used to specify SACM executable assurance case patterns. The **assurance case editor tool** ACEditor provides support for the creation of these constraints within SACM-compliant pattern models. The **model-driven methodology** proposes a two-phase process for the specification and instantiation of these patterns. In the first phase, the executable assurance case patterns are specified within the ACEditor tool with constraints sub-types and computer language queries. In the second phase, the safety analysis is performed and the fault tree result model is transformed into an Open Dependability Exchange (ODE) model which is then used in the automatic instantiation implemented by the **instantiation algorithm**. The feasibility of the proposed **model-driven methodology** in supporting the automatic syntheses of assurance cases was **evaluated** in two systems from the automotive domain. The efficiency has been proved by the low time to instantiate the assurance case patterns. Humans are not able to manually instantiate the evaluated assurance case patterns and obtain the same results. However, the efficiency of the automatic instantiation in runtime assurance and real-time application scenarios must be further explored. The effectiveness has been proved by the high number of elements correctly instantiated and the low number of omissions within the generated product assurance cases. Therefore, the proposed methodology has been analyzed confirming its efficiency and effectiveness in real-world CPS.

The proposed methodology enhances traceability between assurance cases and system models, supports the creation of executable assurance cases, and improves the

semantics of ImplementationConstraint elements. This methodology also provides clear guidelines for specifying executable assurance case patterns, thereby reducing errors in this phase. The Assurance Case Editor (ACEditor) tool increases the usability for generating SACM-compliant patterns. Additionally, the automatic instantiation by the instantiation algorithm significantly reduces the time, cost, and errors associated with generating product assurance cases from patterns. The methodology has been proven applicable to CPS systems, advancing the field of system assurance and contributing to runtime assurance of CPS.

Tool limitations: *i)* the novel model-driven methodology and the use of the ACEditor tool may require a steep learning curve for practitioners, potentially impacting initial adoption and productivity; *ii)* experimental studies in the industry still need to be conducted to evaluate the usability of both SACM visual notation and assurance case modeling tools in supporting the specification of executable argument patterns enriched with ImplementationConstraints; *iii)* the transformation algorithm developed to convert the input fault tree models into ODE-compliant models does not support third-party MBSA tools other than HiP-HOPs;

Instantiation limitations: *i)* the methodology has been validated in specific case studies, but its scalability to larger and more diverse assurance case patterns remains to be thoroughly tested; *ii)* the methodology has primarily been tested in the automotive domain, and its applicability to other domains with different requirements and constraints needs further exploration; *iii)* the lack of support for simplifying the specification of complex model queries involving references to elements from different external artifacts.

Future work: *i)* exploit ontologies to simplify the specification of complex model queries within SACM executable assurance case patterns; *ii)* fulfill the lack of support for third-party MBSA tools to convert the input fault tree models into ODE-compliant models; *iii)* conduct an experimental study to evaluate the usability of both SACM visual notation and assurance case modeling tools in supporting the specification of executable argument patterns enriched with ImplementationConstraints; *iv)* application of the proposed methodology in larger and more diverse systems of different domains using different assurance case patterns. *v)* integrate assurance cases with runtime monitors such as SAFE-ML to support the update of assurance at runtime based on the data of runtime monitors. *vi)* integrate assurance cases with digital twins (i.e., a digital copy of the system) to enable the evaluation of assurance claims based on digital twins data at runtime.

REFERENCES

ACWG. **Goal Structuring Notation Community Standard (Version 3)**. SCSC, 2022. Available at: <https://scsc.uk/scsc-141C>. Access on: January 12^{th}, 2023.

ASLANSEFAT, Koorosh; NIKOLAOU, Panagiota; WALKER, Martin; AKRAM, Mohammed Naveed; SOROKOS, Ioannis; REICH, Jan; KOLIOS, Panayiotis; MICHAEL, Maria K.; THEOCHARIDES, Theocharis; ELLINAS, Georgios; SCHNEIDER, Daniel; PAPADOPOULOS, Yiannis. SafeDrones: Real-Time Reliability Evaluation of UAVs Using Executable Digital Dependable Identities. In: MODEL-BASED Safety and Assessment: 8th International Symposium, IMBSA 2022, Munich, Germany, September 5–7, 2022, Proceedings. Munich, Germany: Springer-Verlag, 2022. P. 252–266. ISBN 978-3-031-15841-4. DOI: 10.1007/978-3-031-15842-1_18. Available from: <https://doi.org/10.1007/978-3-031-15842-1_18>.

ATKINSON, Colin; KUHNE, Thomas. Model-driven development: a metamodeling foundation. **IEEE software**, IEEE, v. 20, n. 5, p. 36–41, 2003.

AVIZIENIS, Algirdas; LAPRIE, Jean-Claude; RANDELL, Brian, et al. Fundamental concepts of dependability. **Technical Report Series-University of Newcastle upon Tyne Computing Science**, University of Newcastle upon Tyne, 2001.

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

BLOOMFIELD, R.; BISHOP, P. Safety and Assurance Cases: Past, Present and Possible Future - an Adelard Perspective. In: DALE, C.; ANDERSON, T. (Eds.), p. 51–67.

BLOOMFIELD, Robin; BISHOP, Peter. Safety and assurance cases: Past, present and possible future–an Adelard perspective. In: SPRINGER. MAKING Systems Safer: Proceedings of the Eighteenth Safety-Critical Systems Symposium, Bristol, UK, 9-11th February 2010. 2009. P. 51–67.

BROWN, Simon. Overview of IEC 61508 Design of electrical/electronic/programmable electronic safety-related systems. **Computing and Control Engineering Journal**, Citeseer, v. 11, n. 1, p. 6–12, 2000.

CHESS. **Composition with guarantees for High-integrity Embedded Software components aSsembly**. Available at: <http://www.chess-project.org>. Access on: Nov 15^{th}, 2023.

CMU-SEI. **EMFTA: EMF-based Fault-Tree Analysis Tool**. Available at: <https://github.com/cmu-sei/emfta>. Access on: Jun 20^{th}, 2023.

DAJSUREN, Yanja; BRAND, Mark van den. **Automotive Systems and Software Engineering: State of the Art and Future Trends**. 1st: Springer Publishing Company, Incorporated, 2019. ISBN 3030121569.

DEIS. **D3.1: Digital Dependability Identities and the ODE Meta-Model**. 2020. Available at: <https://deis-project.eu/dissemination/>. Accessed on: January 12^{th}, 2023.

DELANGE, Julien; FEILER, Peter H. Supporting the ARP4761 safety assessment process with AADL. In: EMBEDDED real time software and systems (ERTS2014). 2014.

DENNEY, E.; PAI, G. A Formal Basis for Safety Case Patterns. In: BITSCH, F.; GUIOCHET, J.; KAÂNICHE, M. (Eds.). **Computer Safety, Reliability, and Security**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. P. 21–32. ISBN 978-3-642-40793-2.

DENNEY, Ewen; PAI, Ganesh. Evidence arguments for using formal methods in software certification. In: IEEE. SOFTWARE Reliability Engineering Workshops (ISSREW), 2013 IEEE International Symposium on. 2013. P. 375–380.

ECLIPSE. **Eclipse Modeling Framework (EMF)**. 2018. Available at: <http://www.eclipse.org/modeling/emf/>. Access on: January 4^{th}, 2023.

ECLIPSE. **Epsilon**. 2022. Available at: <https://www.eclipse.org/epsilon/>. Access on: January 12^{th}, 2023.

ECLIPSE. **Graphical Modeling Project (GMP)**. 2018. Available at: <http://www.eclipse.org/modeling/gmp/>. Access on: January 4^{th}, 2023.

GSN. **GSN Community Standard Version 3 (2022)**. 2018. Available at: <https://scsc.uk/r141B:1?t=1>. Access on: January 4^{th}, 2023.

HABLI, I; KELLY, T. A Safety Case Approach to Assuring Configurable Architectures of Safety-Critical Product Lines. In: GIESE, H. (Ed.). **Archi. Crit. Sys., Inter. Symp., ISARCS 2010, Prague, Proceedings**. Springer, 2010. v. 6150. (LNCS), p. 142–160.

HAWKINS, R. D.; KELLY, T. A Systematic Approach for Developing Software Safety Arguments. English. In: 27TH International System Safety Conference. July 2010. P. 25–33.

HAWKINS, Richard; HABLI, Ibrahim; KOLOVOS, Dimitris; PAIGE, Richard; KELLY, Tim. Weaving an assurance case from design: a model-based approach. In: IEEE. HIGH Assurance Systems Engineering (HASE), 2015 IEEE 16th International Symposium on. 2015. P. 110–117.

ISO. ISO 26262: Road Vehicles Functional Safety, 2018.

JOHNSON, Leslie A et al. DO-178B, Software considerations in airborne systems and equipment certification. **Crosstalk, October**, v. 199, 1998.

JOSHI, Anjali; HEIMDAHL, Mats PE; MILLER, Steven P; WHALEN, Mike W. **Model-based safety analysis**. 2006. NASA Techreport.

KELLY, Tim; WEAVER, Rob. The goal structuring notation–a safety argument notation. In: CITESEER. PROCEEDINGS of the dependable systems and networks 2004 workshop on assurance cases. 2004. P. 6.

KELLY, Tim P; MCDERMID, John A. Safety case construction and reuse using patterns. In: SAFE Comp 97. Springer, 1997. P. 55–69.

KOLOVOS, Dimitrios S.; PAIGE, Richard F.; POLACK, Fiona A. C. The Epsilon Object Language (EOL). In: RENSINK, Arend; WARMER, Jos (Eds.). **Model Driven Architecture – Foundations and Applications**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. P. 128–142. ISBN 978-3-540-35910-4.

KOLOVOS, Dimitris; ROSE, Louis; GARCÍA-DOMÍNGUEZ, Antonio; PAIGE, Richard. **The Epsilon Book**. 2013. Available at: <https://www.eclipse.org/epsilon/doc/book/>. Access on: January 12^{th}, 2023.

KOPETZ, Hermann; BONDAVALLI, Andrea; BRANCATI, Francesco; FRÖMEL, Bernhard; HÖFTBERGER, Oliver; IACOB, Sorin. Emergence in Cyber-Physical Systems-of-Systems (CPSoSs). In: **Cyber-Physical Systems of Systems: Foundations – A Conceptual Model and Some Derivations: The AMADEOS Legacy**. Ed. by Andrea Bondavalli, Sara Bouchenak and Hermann Kopetz. Springer, 2016. P. 73–96. ISBN 978-3-319-47590-5. DOI: 10.1007/978-3-319-47590-5_3. Available from: <https://doi.org/10.1007/978-3-319-47590-5_3>.

MUNK, Peter; NORDMANN, Arne. Model-based safety assessment with SysML and component fault trees: application and lessons learned. **Software and Systems Modeling**, Springer, v. 19, n. 4, p. 889–910, 2020.

NASA. **Fault Tree Analysis Handbook for Aerospace Applications**. 2002. WA, USA.

NASCIMENTO, L. F. A. **SACM: Editor: an OMG standard compliant model-based tool for specification of Assurance Cases for Safety-Critical Systems**. 2020. Course Conclusion Work (Bachelor's in Computer Science). Universidade Federal de Juiz de Fora - Minas Gerais, Juiz de Fora, 2020. http://monografias.nrc.ice.ufjf.br/tcc-web/exibePdf?id=468.

NASCIMENTO, Luis; OLIVEIRA, André L de; VILLELA, Regina; WEI, Ran; HAWKINS, Richard; KELLY, Tim. Runtime Model-Based Assurance of Open and Adaptive Cyber-Physical Systems. In: SPRINGER. INTERNATIONAL Conference on Advanced Information Networking and Applications. 2023. P. 534–546.

O.M.G. **O.M.G.: OCL Version 2.4, 2014,** https://www.omg.org/spec/OCL/2.4/PDF.

OLIVEIRA, André Luiz de. **A model-based approach to support the systematic reuse and generation of safety artefacts in safety-critical software product line engineering**. 2016. PhD thesis – Universidade de São Paulo.

OMG. **Structured Assurance Case Metamodel (SACM) Version 2.2**. 2021. Available at: <https://www.omg.org/spec/SACM/2.2/About-SACM/>. Access on: January 12^{th}, 2023.

PAPADOPOULOS, Yiannis; WALKER, Martin; PARKER, David; RÜDE, Erich; HAMANN, Rainer; UHLIG, Andreas; GRÄTZ, Uwe; LIEN, Rune. Engineering failure analysis and design optimisation with HiP-HOPS. **Engineering Failure Analysis**, Elsevier, v. 18, n. 2, p. 590–608, 2011.

R. DE CASTRO, R.E. Araújo; FREITAS, D. Hybrid ABS with Electric Motor and Friction Brakes. In: PROC. of 22nd Inte. Symp. on Dyna. of Vehi. on Roads and Tracks, (IAVSD11). 2011. P. 1–7.

REICH, Jan; ZELLER, Marc; SCHNEIDER, Daniel. Automated evidence analysis of safety arguments using digital dependability identities. In: SPRINGER. COMPUTER Safety, Reliability, and Security: 38th International Conference, SAFECOMP 2019, Turku, Finland, September 11–13, 2019, Proceedings 38. 2019. P. 254–268.

SELIC, Bran. The pragmatics of model-driven development. **IEEE software**, IEEE, v. 20, n. 5, p. 19–25, 2003.

SELVIANDRO, N.; HAWKINS, R.; HABLI, I. A Visual Notation for the Representation of Assurance Cases Using SACM. In: ZELLER, M.; HÖFIG, K. (Eds.). **Model-Based Safety and Assessment**. Cham: Springer International Publishing, 2020. P. 3–18. ISBN 978-3-030-58920-2.

TRAPP, Mario; SCHNEIDER, Daniel; LIGGESMEYER, Peter. A safety roadmap to cyber-physical systems. In: PERSPECTIVES on the future of software engineering. Springer, 2013. P. 81–94.

TRIBBLE, Alan C; LEMPIA, David L; MILLER, Steven P. Software safety analysis of a flight guidance system. In: IEEE. PROCEEDINGS. The 21st Digital Avionics Systems Conference. 2002. v. 2, p. 13c1–13c1.

WEAVER, Robert Andrew. **The safety of software: Constructing and assuring arguments**. 2003. PhD thesis – University of York, Department of Computer Science.

WEI, R.; KELLY, T. P.; DAI, X.; ZHAO, S.; HAWKINS, R. Model based system assurance using the structured assurance case metamodel. **J. of Syst. and Soft.**, v. 154, p. 211–233, 2019.

WEI, R.; KELLY, T. P.; HAWKINS, R.; ARMENGAUD, E. DEIS: Dependability Engineering Innovation for Cyber-Physical Systems. In: SEIDL, M.; ZSCHALER, S. (Eds.). **Software Technologies: Applications and Foundations - STAF 2017 Collocated Workshops, Marburg, Germany, July 17-21**. Springer, 2017. v. 10748. (Lecture Notes in Computer Science), p. 409–416.

WEI, Ran; KELLY, Tim; REICH, Jan; GERASIMOU, Simos. On the Transition from Design Time to Runtime Model-Based Assurance Cases. In: MODELS (Workshops). 2018. P. 56–61.

WEI, Ran; KELLY, Tim P; DAI, Xiaotian; ZHAO, Shuai; HAWKINS, Richard. Model Based System Assurance Using the Structured Assurance Case Metamodel. **Journal of Systems and Software**, Elsevier, 2019.

WEI, Ran; KELLY, Tim P; HAWKINS, Richard; ARMENGAUD, Eric. Deis: Dependability engineering innovation for cyber-physical systems. In: SPRINGER. SOFTWARE Technologies: Applications and Foundations: STAF 2017 Collocated Workshops, Marburg, Germany, July 17-21, 2017, Revised Selected Papers. 2018. P. 409–416.

ZELLER, Marc; SOROKOS, Ioannis; REICH, Jan; ADLER, Rasmus; SCHNEIDER, Daniel. Open dependability exchange metamodel: a format to exchange safety information. In: IEEE. 2023 Annual Reliability and Maintainability Symposium (RAMS). 2023. P. 1–7.