The Federal University of Juiz de Fora
Exact Sciences Institute
Bachelor Degree in Computer Science

SACM Editor: an OMG standard compliant model-based tool for specification of Assurance Cases for Safety-Critical Systems

Luis Felipe de Almeida Nascimento

JUIZ DE FORA
NOVEMBER, 2019

SACM Editor: an OMG standard compliant model-based tool for specification of Assurance Cases for Safety-Critical Systems

Luis Felipe de Almeida Nascimento

The Federal University of Juiz de Fora

Exact Sciences Institute

Department of Computer Science

Bachelor Degree in Computer Science

Orientador: André Luiz de Oliveira

JUIZ DE FORA

NOVEMBER, 2019

SACM Editor: an OMG standard compliant model-based tool for specification of Assurance Cases for Safety-Critical Systems

Luis Felipe de Almeida Nascimento

MONOGRAPHY SUBMITTED TO THE FACULTY OF THE EXACT SCIENCES INSTITUTE OF THE FEDERAL UNIVERSITY OF JUIZ DE FORA, AS AN INTEGRAL PART OF THE REQUIREMENTS NECESSARY TO ATTAIN A BACHELOR'S DEGREE IN COMPUTER SCIENCE.

Approved by:

André Luiz de Oliveira
Ph.D in Computer Science

Gleiph Ghiotto Lima de Menezes
Ph.D in Computer Science

Ciro de Barros Barbosa
Ph.D in Computer Science

JUIZ DE FORA

14 DE NOVEMBER, 2019

Abstract

Critical systems are systems whose failures may result in death or serious damages to finances, property or the system environment. Due to their critical nature, these systems should be developed following guidelines prescribed safety standards. Standards demand that safety properties should be analyzed and verified at different levels of abstraction. At the requirements, potential threats to the system safety should be identified and mitigated. At architectural design, engineers should analyze how system failure propagates through subsystems, and later identify how components contribute to system failures. Standards from the automotive and aerospace domains also recommend or require the specification of an assurance case as a requirement for achieving certification. Assurance case or safety case provides a way to argue why the system is acceptably safe to operate a system within a determined context, supported by a body of evidence. Goal Structured notation (GSN) and Structured Assurance Case Metamodel (SACM) are graphical notations that support the specification of an assurance case. Model-driven development has been widely adopted in the development and assurance of critical systems. Nowadays there is a lack of tools that provide support for safety engineers to specify assurance cases according to OMG-SACM standard and performing models transformation from GSN to SACM notations. In order to fill this gap, this project proposes the development of a tool that supports the specification of assurance case using SACM notation, and it provides GSN2SAMC model-transformations to support compatibility between both assurance case modeling notations. In this work, a graphical editor has been developed upon the Eclipse Modeling Framework (EMF) and Graphical Modeling Framework (GMF) platforms. The tool has been validated in two case studies, one in the automotive domain, and the other in the aerospace domain. The developed SACM Editor contributed to support users in the specification of SACM compliant assurance cases, reducing the time and costs of this task.

Keywords: Assurance Case, Safety Case, Structured Assurance Case Metamodel.

Resumo

Sistemas críticos são sistemas onde falhas podem resultar em mortes ou danos sérios como econômicos, à propriedade ou ao ambiente do sistema. Devido a sua natureza crítica, esses sistemas devem ser desenvolvidos de acordo com as normas prescritas nos padrões de segurança, tais padrões exigem que as propriedades de segurança seja mostradas em diferentes níveis de abstração, como identificar ameaças potenciais à segurança e mitigar-las no nível de requisitos, e no arquitetural analisar a propagação de falhas através dos sub-sistemas e identificar a contribuição de cada componente. Os padrões do domínio automotivo e aeroespacial recomendam ou exigem a especificação de um caso de garantia para obter a certificação. Um caso de garantia ou caso de segurança fornece um modo de argumentar sobre os motivos do sistema ser seguro para operar em um contexto especifico, apoiado-o por um corpo de evidências. Goal Structured notation (GSN) e Structured Assurance Case Metamodel (SACM) são anotações gráficas que suportam a especificação de um caso de garantia. O desenvolvimento orientado a modelos foi amplamente adotado no desenvolvimento e garantia de sistemas críticos. Atualmente, existe uma falta de ferramentas que ofereçam suporte a especificação de caso de garantia de acordo com o padrão OMG-SACM e transformação do modelo das notações GSN para SACM. Para preencher essa lacuna, este projeto propõe o desenvolvimento de uma ferramenta que suporte a especificação de caso de garantia usando a notação SACM e fornece transformações de modelo GSN2SAMC para suportar a compatibilidade entre as duas notações de modelagem de caso de garantia. Neste trabalho, um editor gráfico foi desenvolvido nas plataformas EclipseModeling Framework (EMF) e Graphical Modeling Framework (GMF). A ferramenta foi validada em dois estudos de caso, um no domínio automotivo e outro no domínio aeroespacial. O editor SACM desenvolvido contribuiu para apoiar especificação casos de garantia compatíveis com SACM e reduziu o custo para faze-la.

Keywords: Caso de Garantia, Caso de Segurança, SACM, SACM 2.1, Editor.

Contents

List of Figures

List of Tables

List of abbreviations

ARP	Aerospace Recommended Practice
CAE	Claim Argument Evidence
EMF	Eclipse Modeling Framework
Epsilon	Extensible Platform of Integrated Languages for mOdel maNagement
GEF	Graphical Editing Framework
GMF	Graphical Modeling Framework
GSN	Goal Structured Notation
HBS	Hybrid Braking System
ISO	International Organization for Standardization
MDE	Model-Driven Engineering
MDD	Model-Driven Development
MOF	Meta Object Facility
SACM	Structured Assurance Case Metamodel
SAE	Society of Automotive Engineers
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	eXtensible Markup Language

1 Introduction

Critical systems can range from small devices to complex systems of industrial process management. Sommerville (2003) considers critical systems as those whose failures can result in economic loss, physical damage or threats to human life. The critical nature of these systems demands that they must address availability, reliability, safety and security requirements. The achievement of these requirements can be demonstrated by performing safety engineering activities.

Safety engineering comprises a set of activities that must be performed in parallel to development activities. Hazard identification, risk assessment, and allocation of safety requirements are examples of safety engineering activities. In order to obtain certification for a critical system, safety standards recommend or require the specification of an assurance case in addition to safety engineering activities (SAE, 2010).

For the development of critical systems, it is necessary to follow guidelines established in safety standards, e.g., ISO 26262 for automotive domain and SAE ARP 4754A aircraft domain. Safety standards establish the safety properties of a critical system should be analyzed and demonstrated at different levels of abstraction, from requirements to a 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. Standards also require or recommend the development of an assurance case/safety case as a requirement for safety certification.

An assurance case is a clear, comprehensive and defensible argument, supported by a body of evidence, that a system is acceptably safe to operate in a particular context (KELLY; WEAVER, 2004). There exists in the literature textual, tabular and graphical notations for specifying an assurance case. Initially, an assurance case was specified in a free text notation. The usage of natural language may ambiguity in the argument. In order to improve representation of assurance cases, tabular and graphical notations have been created (KELLY; MCDERMID, 1997). Goal Structured Notation (GSN), Claim Argument Evidence (CAE), and Structured Assurance Case Metamodel (SACM) are ex-

amples of graphical notations that support the specification of assurance cases.

Model-Driven Engineering techniques provide the benefits of a clear expression of requirements and architecture and automation. Model-based techniques have been widely adopted in the development and assurance of critical systems. Eclipse Modeling Framework (EMF) (ECLIPSE, 2018a) and Graphical Modeling Framework (GMF) (ECLIPSE, 2018c) are examples of model-based techniques and tools. The EMF is built upon the Meta-Object Facility (MOF) standard. Model-based techniques can be used to support for specification of assurance cases in GSN (GSN, 2018) and SACM (OMG, 2019) metamodel.

1.1 Problem

The SACM is an OMG specification that defines the requirements for assurance case modeling notations. The GSN is a SACM-compliant modeling notation. In addition to GSN, SACM metamodel provides support for specifying the provenance of evidence items that support assurance claims. Although SACM metamodel defines a standardization to be followed by safety argumentation, i.e., assurance cases, there is a lack of tooling support for specifying assurance cases in SACM notation. In order to fill this gap, in this final project is proposed SACM Assurance Case Editor (SACM ACEditor), built upon EMF platform and GMF, to support engineers on the specification of SACM-compliant assurance cases for critical systems. SACM ACEditor provides a graphical user interface to aid safety engineers specifying assurance cases and assurance case patterns. The proposed editor intends to contribute to reducing the effort of engineers in the production of assurance cases for critical systems, required by standards and authorities as a requirement for certification and release for operation.

1.2 Objectives

The main goals of this work are: *i)* the development of an EMF-compliant graphical editor to support the specification of assurance cases in compliance with OMG SACM 2.1 meta-model; *ii)* the definition of a process to support engineers in the development

of modeling tools based on EMF and GMF platforms; *iii)* the provision of compatibility support between GSN and SACM via GSN2SACM model transformations, which enables the conversion of a GSN model to an equivalent representation in SACM; and iv) providing the validation of the proposed SACM ACEditor through demonstrating its usage in two realistic case studies, carried out in the automotive and aerospace domains.

1.3 Results

The results of this final project are the SACM ACEditor, an assurance case editor developed upon Eclipse platform that supports the specification of assurance cases according to the SACM 2.1 meta-model, and the transformation of GSN models into SACM. Both SACM ACEditor and the developed model transformation have been validated on the automotive Hybrid Braking System and Tiriba unmanned aircraft Flight Control System.

1.4 Organization

The next chapters are briefly described. **Chapter 2** presents all the concepts and knowledge needed for contextualization and a better understanding of this work. This chapter explains the Model-Driven Engineering concept, the tools used to develop the editor, the basic concepts of safety engineering and safety standards, the definition of assurance case and assurance case pattern, the GSN notation and SACM 2.1 meta-model, and finally an overview of automatic instantiation of assurance case patterns. **Chapter 3** describes the development process followed by the author to build the SACM ACEditor in a generic way, for supporting engineers in the construction of EMF model editors for other domain-specific modeling languages. **Chapter 4** presents the architecture of the proposed SACM Assurance Case Editor (SACM ACEditor). **Chapter 5** describes the assurance case patterns used in case studies. **Chapter 6** describes the usage of SACM ACEditor in the specification of assurance cases for an automotive braking system. Finally, **Chapter 7** presents an analysis of the contributions and limitations of this work and future research directions.

2 Background

This chapter presents the background concepts needed for the reading understanding the context of the research contributions. Section 2.1 describes the concepts of Model-Driven Development. Section 2.2 presents an overview of Eclipse Model Framework and Graphical Modeling Framework platform used in the development of SACM ACEditor. Section 2.3 describes the basic concepts of safety engineering and provides a brief overview of safety standards. Section 2.4 contains Assurance Case and Assurance Case Pattern definitions. Goal Structuring Notation (GSN) is detailed in Section 2.5. Section 2.6 provides an overview of OMG Structured Assurance Case Metamodel. Finally, in Section 2.7 is presented a discussion on automatic instantiation of assurance case patterns.

2.1 Model-Driven Engineering

Model-Driven Engineering (MDE) or Model-Driven Development (MDD) aims to make the artifacts useful to their particular propose, at their specific stage of the life cycle, e.g., to describe architecture for the underlying need to link related artifacts to it and to serve as a means of communication for all those participating in the project, (HAILPERN; TARR, 2006). MDE focuses on models as the first-class entity of a software development process rather than computer programs, because, models can guide the software development. A model is an abstraction with an intended and defined purpose (SELIC, 2003). A MDE approach includes Domain-Specific Modeling and Model Management.

The Domain-Specific Modeling allows domain experts capturing 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 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, comparison, 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, 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 Unified Modeling Language (UML) is an example of a model-based technique that provides a standard visual language for systems modeling. The UML supports the specification of the system at different levels of abstraction, e.g., requirements and detailed design. Thus, at high levels, it allows the abstraction of a large part of the necessary technology and implementation needed for system development. The existing UML supports automated change impact analysis, model transformation, and generation of source code from an UML model (MELLOR; CLARK; FUTAGAMI, 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 2.1 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**, e.g., the SACM 2.1 meta-model specification. **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).

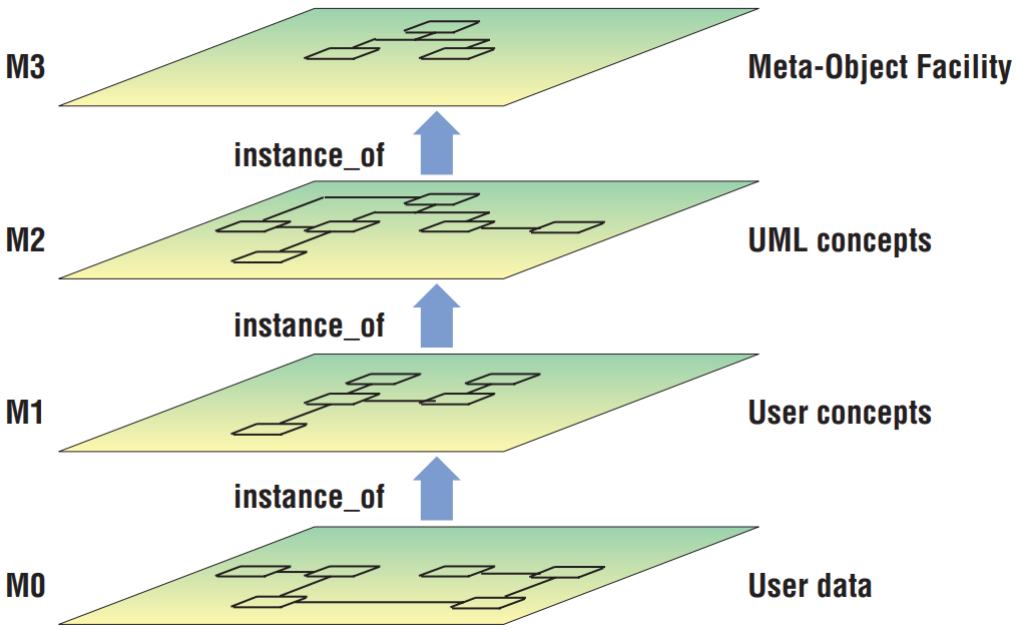

Figure 2.1: OMG MOF infrastructure, (ATKINSON; KUHNE, 2003)

Structured meta-models based on the MOF infrastructure provide automated support for model verification, validation, transformation, merging, and comparison (KOLOVOS et al., 2013). In the Eclipse Modeling Framework (EMF), a MOF-compliant platform, these tasks are supported by **Ecore** meta-modeling language. The **Ecore** is built upon object-oriented concepts of classes and inheritance, used to specify modeling languages within the EMF platform with the support of customized model editors.

2.2 Modeling Tools

This section presents the modeling techniques and tools that have been used in the development of this project. It is presented an overview of Eclipse Modeling Framework (EMF), Graphical Modeling Framework (GMF), the Epsilon model management languages and EuGENia tools. This section describes how developing a graphical model editor for a given domain-specific language using these tools.

2.2.1 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) is a MOF-compliant modeling platform. The EMF platform supports the automated code generation from a meta-model specification. EMF unifies java, XML and UML, allowing models to be defined in one of these formats. The EMF also supports the generation models from other models. Modeling and programming in the EMF platform can be considered the same thing, it is not necessary to choose between one and another, (STEINBERG et al., 2008). Modeling supports engineers in identifying what the system should do more easily than using only the source-code.

Figure 2.2, (ECLIPSE, 2018a) illustrates the structure of the EMF platform, which comprises: the specification of a domain-specific model upon the Ecore meta-model, EMF.edit, and EFM.editor source code automatically generated from an Ecore domain-specific model. The EMF.edit provides generic classes to support the EMF.editor. Both EMF.edit and EMF.editor source code are automatically generated via execution of EMF.Codegen capability. The Ecore is an object-oriented meta-modeling language from EMF for producing the concrete artifacts, e.g., code and configuration files from which a model editor is built.

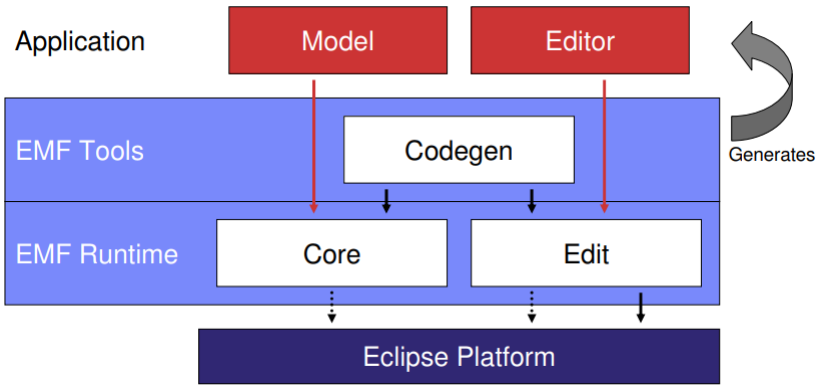

Figure 2.2: Architecture of EMF

2.2.2 Epsilon

The **E**xtensible **P**latform of **I**ntegrated **L**anguages for m**O**del ma**N**agement (Epsilon) is a family of languages and tools for code generation, transformation, validation, comparison, migration and refactoring of models. Epsilon can be used to manipulate EMF and other types of models, (KOLOVOS et al., 2013). The current Epsilon version provides

the following languages: EOL, EVL, ETL, ECL, EML, EWL, EGL and EPL. A brief description of each one of these languages is presented in the following:

- Epsilon Object Language (EOL): provides a reusable set of common model management facilities. EOL can be used for automating tasks that do not fall into the patterns targeted by other Epsilon languages, e.g., automatic fixes in a model generated using EuGENia tool before code generation.

- Epsilon Transformation Language (ETL): used to transform an arbitrary number of input models into an arbitrary number of output models. ETL allows the specification of transformations from GSN models to SACM models.

- Epsilon Comparison Language (ECL): it supports the specification of comparison algorithms in a *rule-based* manner to identify pairs of matching elements between two models from potentially different meta-models and modeling technologies.

- Epsilon Merging Language (EML): it supports merging an arbitrary number of input models from potentially diverse meta-models and modeling technologies.

- Epsilon Wizard Language (EWL): it supports the specification of small update model transformations in a rule-based manner. This kind of transformation performs *in-place* modifications in the source model itself. This capability is not provided by ETL and cannot be implemented using EOL.

- Epsilon Generation Language (EGL): is a template-based code generator. An EGL program resembles the text that they generate, and it provides features that simplify and support the generation of text-from-model transformations. EGL can be used to transform models into various types of textual artifacts, e.g., Java code, and HTML reports.

- Epsilon Pattern Language (EPL): is a pattern matching the language that allows run-time interoperability and reuse of code with languages that support a range of model management tasks. EPL provides support for specifying patterns that involve model elements that conform to different modeling technologies.

2.2.3 Graphical Modeling Framework and EuGENia

The Graphical Modeling Framework (GMF) platform supports the automatic generation of source code for graphical editors for domain-specific languages specified in Ecore models built upon the EMF platform (ECLIPSE, 2018b). In order to generate a graphical editor based on both EMF and GMF platforms, it is necessary defining a meta-model of the domain problem, which includes the domain elements and their relationships using Ecore. The EMF platform provides embedded resources for editing a model. The EuGENia is an Epsilon tool that allows the automatic generation of the EMF.model base classes, EFM.edit, EMF.editor and GMF.diagram source code from a text-based Ecore meta-model specification and the mappings between model elements and their graphical representation using the Emfatic language (ECLIPSE, 2018b).

Epsilon Emfatic is a textual language for the specification of EMF meta-models and mappings linking meta-model elements to their graphical representation. The Emfatic has its own syntax and notation that allows the definition of mappings between `org.eclipse.draw2d.Figure` classes and Ecore model elements. The Emfatic language provides annotations to support linking graphical representations to EMF domain model elements. For example, Emfatic annotations can be used to highlight if a domain model element is a node or a link, and its associated graphical representation, which can be a Java class that implements IFigure interface from `org.eclipse.draw2d` API (ECLIPSE, 2018b). Figure 2.3 illustrates the workflow of automatic generation of GMF and EMF editors using EuGENia, (KOLOVOS et al., 2010; EPSILON, a). However, Two steps have been added 'Emfatic Meta-model Specification' and 'EuGENia Enfatic2EcoreTransformation'.

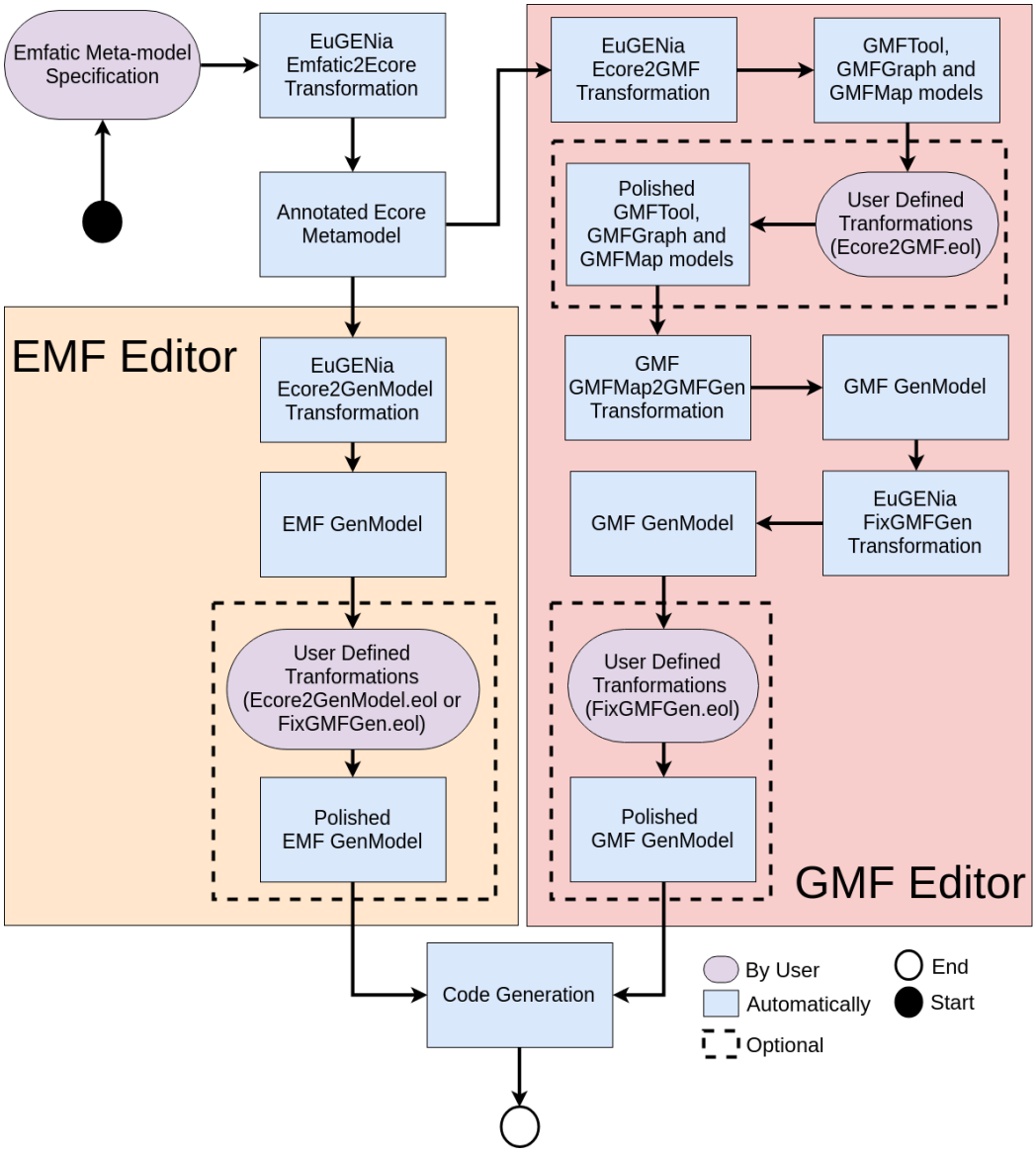

Figure 2.3: Eugenia Workflow for generation of source for EMF and GMF model editors.

As shown in figure 2.3 the first step of the process is specifying a domain-specific language using Emfatic. This specification contains the declaration of the required elements, e.g, classes, and also the definition of Emfatic annotations in each of these elements or in their properties, e.g if the element should be a 'node' or a 'relationship'. After it, the EuGENia tool performs the automatic translation of an Emfatic file into an Annotated Ecore Metamodel. Later, we use EuGENia to generate the generator model for the annotated.Ecore model. Therefore, the model, edit, and editor source code for a basic modeling editor is generated via execution of the EMF generator model associated with the.Ecore file. Additionally, from an annotated. Ecore model, EuGENia automatically generates the GMFTool, GMFGraph and the GMFMap models necessary for creating the

diagram editor for a given domain-specific language specified in an .Ecore model. It is important to highlight that users can define their own Ecore2GMF transformation using Epsilon Object Language (EOL). From the GMFGraph, GMFTool, GMFMap, and user-defined Ecore2GMF transformations, EuGENia supports the generation of the GMF generator model. The execution of the GMF generator model yields the source code of the GMF editor for a domain-specific modeling language.

The following files are needed to build a graphical model editor using EMF and GMF platforms Kolovos et al. (2010):

- **Annotated Ecore Metamodel**: it specifies the abstract syntax of the domain-specific language, the graphical syntax of the language and various implementation options. An.Ecore metamodel is the input file for the EMF and GMF code generators;

- **GMFGraph**: is the graph model that specifies the graphical elements, e.g., shapes, connections, labels, and decorations, which are part of a diagram editor for a given domain-specific language;

- **GMFTool**: specifies the tools for creating model elements to be made available for users in the editor's palette;

- **GMFMap**: this is the mapping model, which provides mappings between graphical elements defined in the graph models and the creation tools, defined in the tooling model with the abstract elements from an Ecore metamodel, e.g., classes, attributes, and references;

- **GMF GenMode**l: provides the generator model file. It provides a more fine-grained transformation of the mapping model (GMFMap), and all the low-level information required by the GMF code generator, producing concrete artifacts. e.g., Java code and configuration files, that generates the graphical editor;

- **EMF GenModel**: an executable generator model that captures lower-level information. It specifies how the meta-model should be implemented in Java, i.e., the Java package under which the code will be generated.

EuGENnia also provides a way to automate fixes and transformations in the Ecore and GMF models. This feature is optional and it can be implemented via execution of EOL programs, or automatically, if they are in the same directory of the models when the EuGENia automatic generation starts. The Ecore2GenModel.eol model transformations support modifications on the Annotated Ecore Metamodel (.Ecore) and EMF GenModel. With the ECore2GMF.eol file it is possible to modify the Annotated Ecore Metamodel, the GMFTool model named GmfTool, the GMFGraph model (GmfGraph) and the GMFMap model (GmfMap). The FixGMFGen.eol transformations allow the Annotated Ecore Metamodel and the GMF GenModel (GmfGen), to be modified, (EPSILON, a).

2.3 Safety Engineering

This section explains the concepts of Safety Engineering and Safety Life-Cycle. This section also describes the relationship between the assurance case with both safety and development life-cycles.

2.3.1 Introduction

Safety Engineering is a discipline that ensures that engineered systems provide acceptable levels of safety. Safety Engineering intends to support the identification and reduction of safety risks to a certain acceptable level, (KANCHANA; FANEY, 2015). Safety engineering comprises a set of activities, e.g., hazard identification, risk assessment and allocation of safety requirements, that must be performed in parallel to development. Safety Engineering should also be executed from the beginning until the end of development.

2.3.2 Safety Life Cycle

The safety life cycle was incorporated into System Engineering to optimize the design and increase system safety. The safety life-cycle comprises three major phases: analysis; realization; and operation. The activities of each phase may vary, depending on the adopted safety standard, (UP-TIME, 2007).

- Analysis: in this phase, the hazards are identified and their risks estimated e.g., using probabilistic attributes such as the likelihood of hazardous events and severity. In this phase, engineers evaluate whether the risks posed by hazards are tolerable to the industry, authorities or regulatory standards.

- Realization: it develops the conceptual design for technology, architecture, periodic test interval, reliability and safety evaluation, as well as the detailed design for installation planning, commissioning, start-up acceptance testing, and design verification.

- Operation: it creates the validation plan, starts the review of operation and maintenance planning. It starts the operation, maintenance, and periodic functional test. This is a phase where modifications and decommissioning can be made.

Safety Engineering activities start from the initial stages of development, where there is the possibility of taking corrective actions to eliminate or minimizing the risks before the final decisions concerning the project being taken (LEVESON, 2003). The hazard analysis and risk assessment, allocation of safety requirements and the provision of safety evidence are the basic activities present in safety engineering. Safety standards cover these basic activities in their safety life-cycle. At the system, during hazard analysis, engineers identify hazardous failure conditions; determine the risk factor associated with each hazardous failure; and allocate safety requirements to eliminate or minimize the failure effects on the overall safety. Safety requirements can be specified in terms of safety integrity levels according to risk tolerability criteria defined in the targeted safety standard.

Safety standards provide guidance on applying methods and techniques to support safety engineering activities, aimed to improve the reliability of the system. They may be advisory or compulsory and are normally laid down by an advisory or regulatory body that may be either voluntary or statutory. However, safety standards can vary according to the targeted domain, industry or even region. For example, there exists safety standards for aerospace, automotive, nuclear and other domains, e.g., ISO 26262 for automotive domain and SAE ARP 4754A for the aerospace domain.

SAE ARP 4754A provides guidelines for the development of aircraft systems taking into account the overall aircraft operating environment, functions, validation of requirements, verification of the design implementation for certification and product assurance. The SAE ARP 4754a also provides best practices for demonstrating compliance with regulations and assisting companies in developing and meeting their own internal standards, (SAE, 2010).

The concept of assurance case has already been considered in different domains such as defense, aerospace, nuclear and railway (KELLY; WEAVER, 2004). Therefore, safety standards from different domains have been recommended or required the specification of an assurance case as a requirement to a system obtain certification credits (OLIVEIRA, 2016).

Safety standards establish that the safety properties of a critical system must be analyzed and demonstrated at different levels of abstraction, i.e., from requirements to components. Thus, safety information can be at different levels of development life-cycle. At the system-level, engineers identify hazards, classify their risks assign safety requirements to minimize hazard effects. At the design, it is needed to analyze how system failures propagate through the system architecture. Finally, at the component level, it is needed to analyze how components can fail and contribute to the occurrence of hazards. Therefore, it is possible to connect the safety life-cycle to the development life-cycle, Figure 2.4 illustrates the relationship between development and safety life-cycles and assurance case. An Assurance case is built upon the evidence provided by development and safety life-cycles.

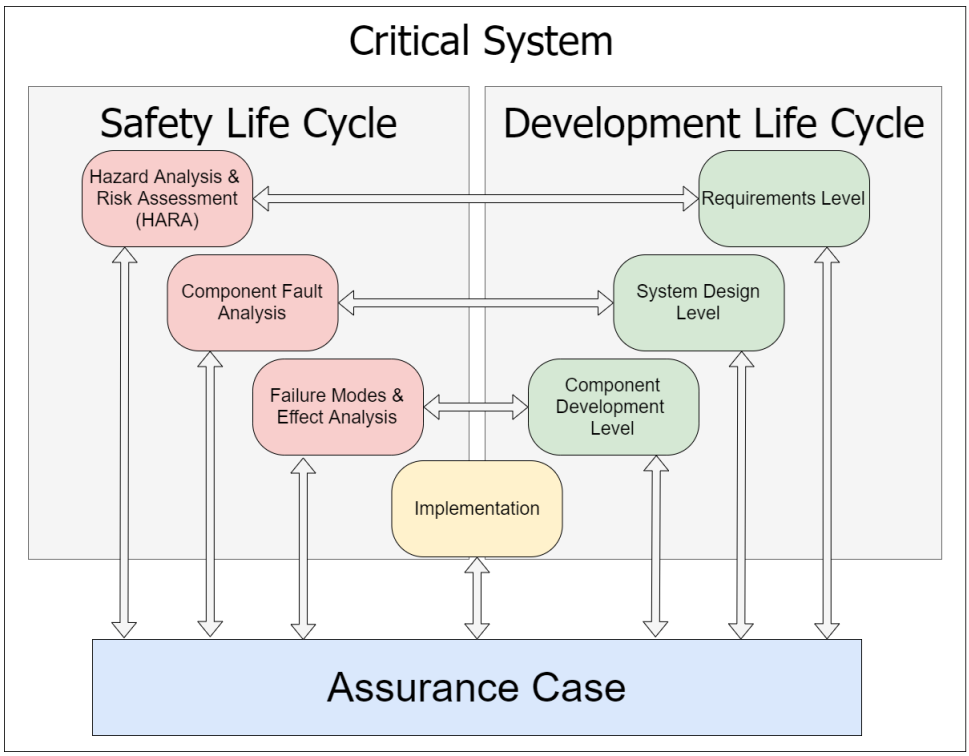

Figure 2.4: Critical System Life Cycle

Nowadays, the development of an assurance case for a critical system is important for supporting and substantiating claims about the safety of a system, or about whether a system has achieved an acceptable level of safety.

2.4 Assurance Case

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**. An assurance is **clear** because 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 absolutely 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 this context, because, no system can be considered safe if it is used in an inappropriate or unexpected manner, (KELLY; WEAVER, 2004).

Several notations have been proposed in the literature to document Assurance Cases. Initially, a free text notation has been created. However, the ambiguity inherent to the natural language makes difficult to accurately expressing 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 in the following shows the limitations of natural language on clearly expressing complex arguments.

> "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).

In order 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. Although graphical notations are not perfect, due to the fact that each notation has its own limitations, they are the best way to represent an Assurance Case. Goal Structured Notation (GSN) and the Structured Assurance Case Metamodel (SACM) are examples of graphical notations that support the specification of assurance cases.

2.4.1 Assurance Case Pattern

Reuse is a software engineering strategy where the process of development is directed to reuse existing software artifacts, e.g., models, requirements, functions (SOMMERVILLE, 2011). Since assurance case is a way to argue the safety of a system, the definition of a pattern must be broadened to reusing existing artifacts of a system.

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. A number of 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. In order 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 a system to another. However, 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 (KELLY; MCDERMID, 1997).

The usage of patterns as a way to document and reuse successful assurance argument structures was developed by Kelly and McDermid (1997). The main objective of an Assurance Case pattern is to provide the reuse of artifacts for the construction of a safety argument. Assurance patterns capture the required form of an assurance argument in a manner that is abstract from the details of a particular argument. It is possible to use these patterns to create specific arguments by instantiating them in a specific system. The information required by instantiating an assurance case pattern can be provided manually from design or documentation analysis, directly from an engineer, or automated (HAWKINS et al., 2015).

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 system 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. All abstract *terms* must be replaced to create/instantiate a concrete assurance case. For example, consider the abstract argument expression "{System X} is acceptably safe". The "System X" is an abstract *term* that must be replaced by the name of a specific system.

Assurance case pattern can be specified in graphical notations such as GSN and SACM 2.1. An Assurance Case pattern represents abstractions from a partial argumentation solution.

2.5 Goal Structured Notation (GSN)

Goal Structured Notation 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 and Weaver (2004).

The GSN base elements are Goal, Solution, Strategy, Context and Undeveloped Goal. Figure 2.5 illustrates the graphical representation of each one of these elements. GSN provides a hierarchical goal structure that decomposes goals into sub-goals. GSN allows engineers demonstrating 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 Fault Tree Analysis (FTA).

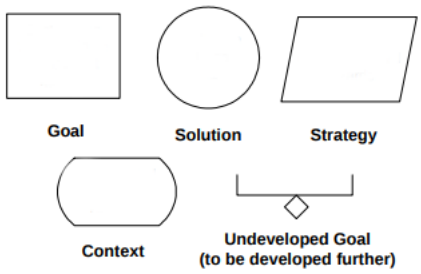

Figure 2.5: Elements of GSN notation

Figure 2.6 shows a 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). The **context** C1 gives the information that the **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).

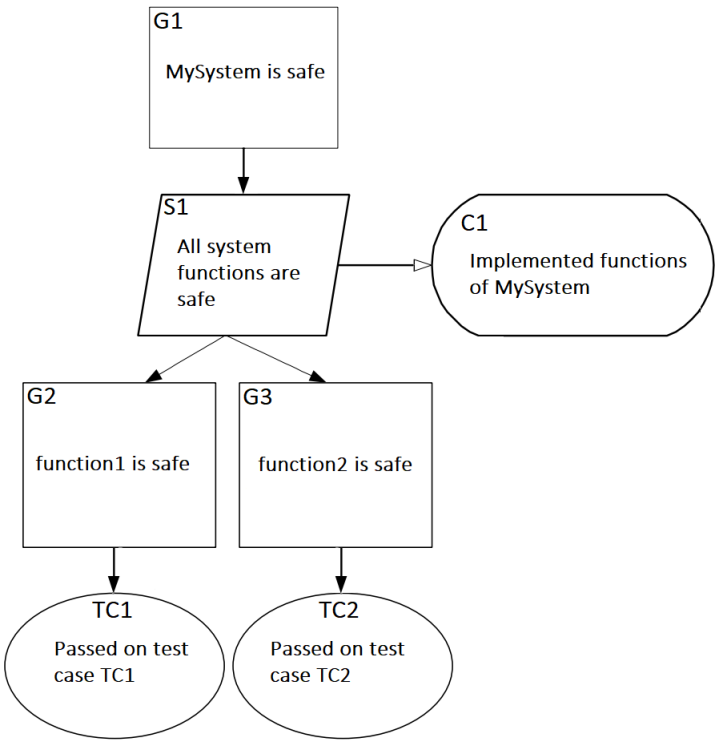

Figure 2.6: An Example of GSN Goal Structure

2.5.1 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 earlier described in the example. The GSN Pattern extension supports two types of abstractions: **structural abstraction**, which supports 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. GSN pattern extension also comprises constraints to represent these abstractions: **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, 2018). GSN **multiplicity**, **optional** and **choice** constraints provide informa-

tion about how to instantiate the elements associated with abstract terms. Figure 2.7 shows these constraints representation and the abstract property representation. Figure 2.8 is a simple and descriptive example of a GSN pattern representation, which has been adapted from Kelly and McDermid (1997).

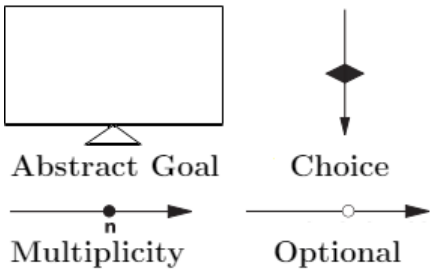

Figure 2.7: GSN Abstractions

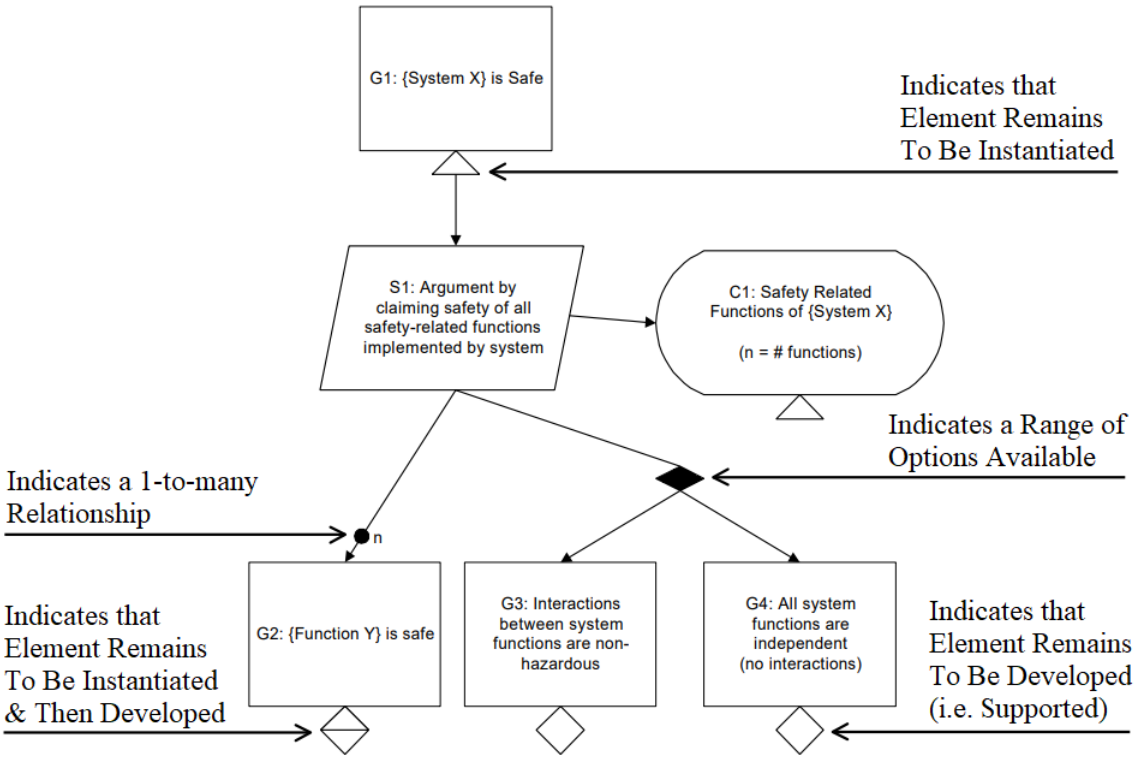

Figure 2.8: GSN Pattern Representation Example

2.6 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 et al., 2019).

SACM provides the following features: modularity; multiple language support; controlled vocabulary; describing the level of trust in arguments; counter-arguments; traceability from evidence to the artifact; automated assurance case instantiation.

The Structured Assurance Case Metamodel version 1.0 was finalized in 2012, and it consists of a top-level object container, merging the Structured Assurance Evidence Metamodel (SAEM), and the Argumentation Metamodel (ARG) without significantly altering the two original meta-models. The current version of SACM is the 2.1, (OMG, 2019).

Based on OMG (2019), the SACM meta-model is divided into different sets of elements. Each set has more specific purposes, and when these sets are grouped together, they compose the SACM 2.1 meta-model illustrated in figure 2.9. The groups are structured into assurance case base classes, assurance case packages, terminology classes, argumentation metamodel, and artifact metamodel.

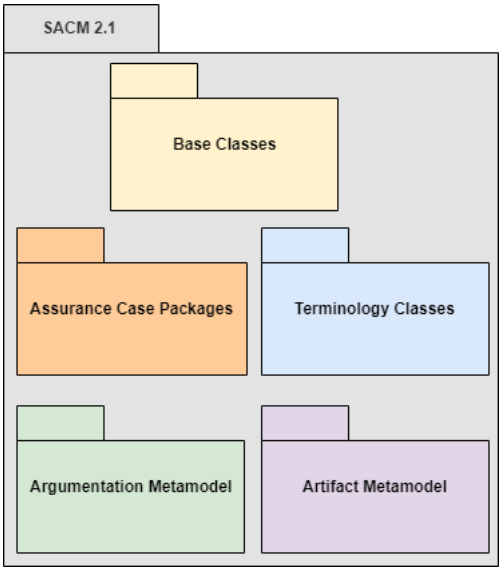

Figure 2.9: Components Of SACM 2.1

2.6.1 Assurance Case Base Classes

Assurance case base classes 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. **ImplementationConstraint** and **Description** are very important for the next sections. **ImplementationConstraint** contains the conditions

that must be fulfilled in order to allow an abstract **ModelElement** become an non-abstract. **Description** is used to provide the 'content' of a **ModelElement**, e.g., it would be used to provide the text of a **Claim**, (OMG, 2019).

The *content* property of **ImplementationConstraint** and **Description** is the **MultilangString** type. This type provides support for multi-languages, i.e., the internationalization of its content. Thus, it is possible to indicate the **ImplementationConstraint** conditions and the 'content' of a **ModelElement** in different languages.

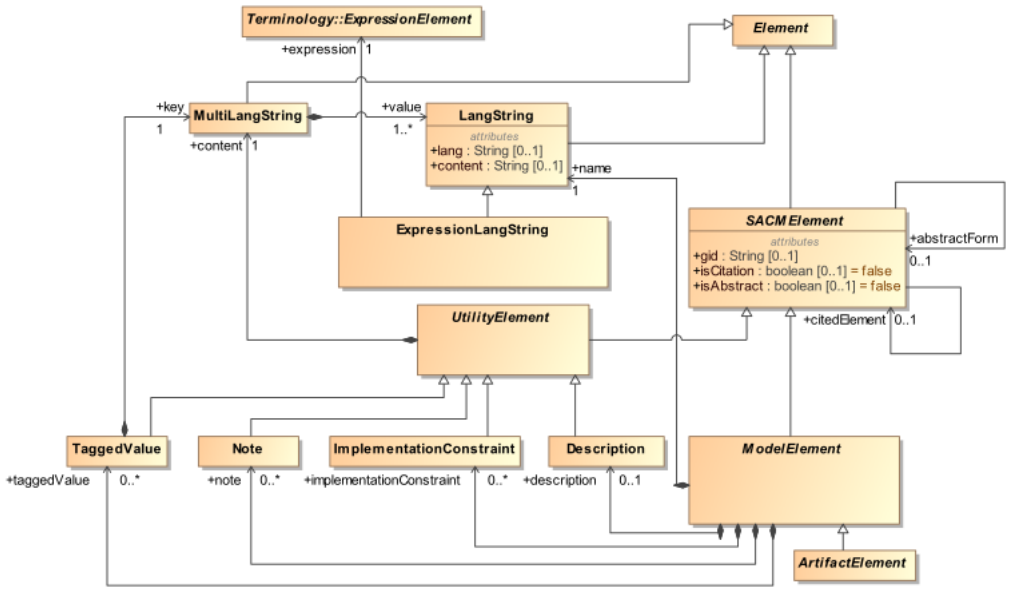

Figure 2.10: SACM 2.1 Assurance Case Base Classes

2.6.2 Structured Assurance Case Packages

Structured assurance case packages allow creating 'modules' that may contain other assurance case packages, including citations to other packages not contained within the same package hierarchy.

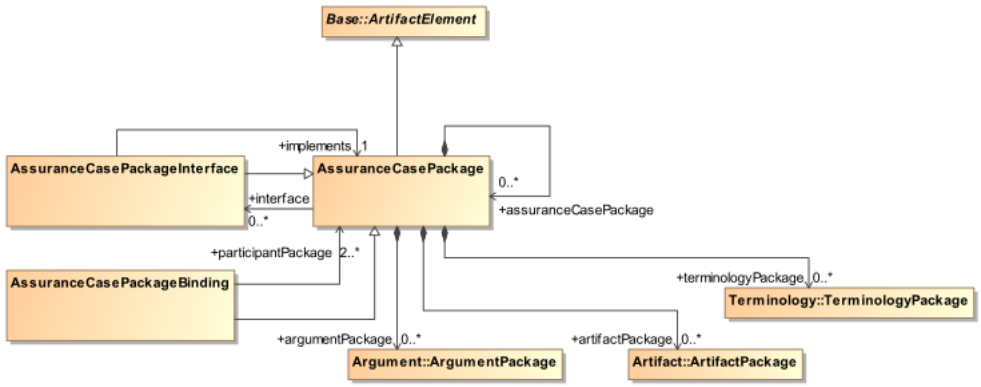

Figure 2.11: SACM 2.1 Structured Assurance Case Packages

2.6.3 Structured Assurance Case Terminology Classes

Structured assurance case terminology classes define the concepts of term and expression, and it provides 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 this abstract **Term** will become a concrete **Term**. The **Expression** is used to construct expressions composed by others **ExpressionElements**, i.e., **Terms** or **Expressions**.

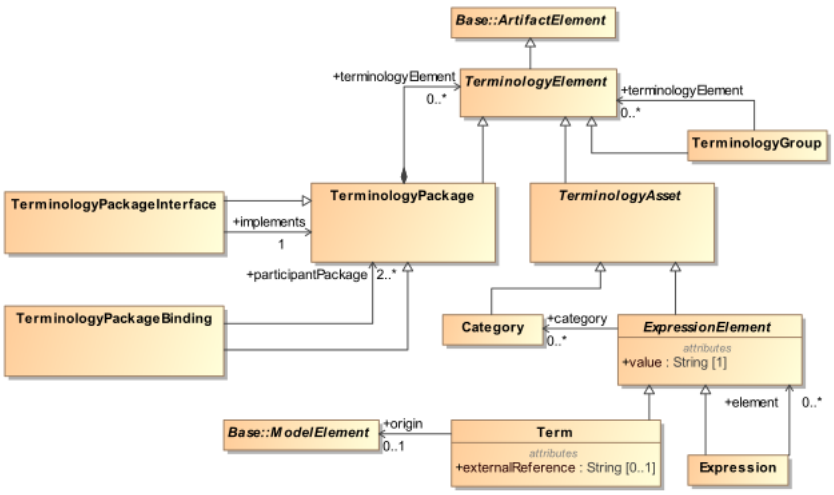

Figure 2.12: SACM 2.1 Structured Assurance Case Terminology Classes

2.6.4 Argumentation Metamodel

Argumentation meta-model defines the necessary concepts to model structured arguments, e.g., elements, relationships among them and their properties.

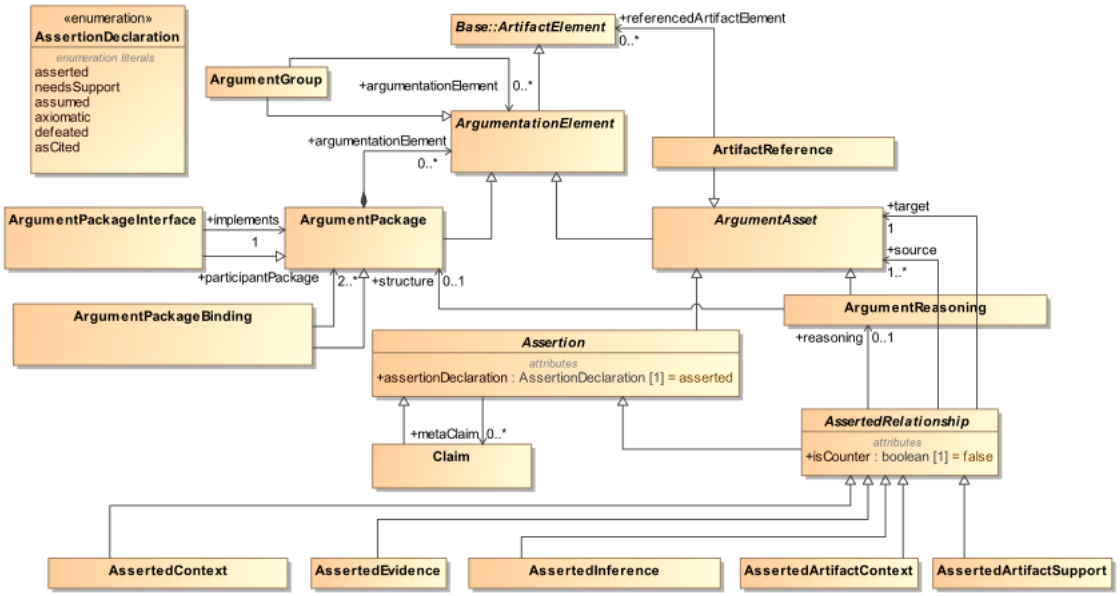

Figure 2.13: SACM 2.1 Argumentation Metamodel

2.6.5 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**, bacause the **ModelElement** extends the **ArtifactElment**, so any **ModelElement** of an assurance case can be in an **ArtifactPackage**. Thus, the definition of **ArtifactPackage** is the broadest within SACM.

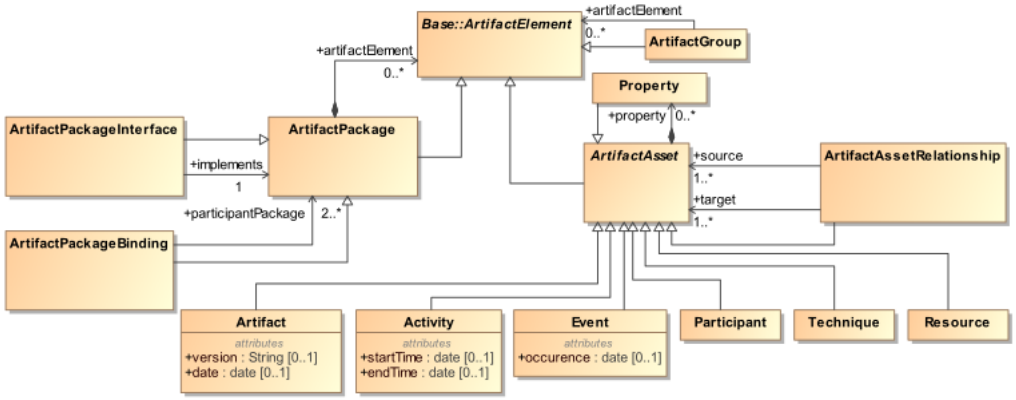

Figure 2.14: SACM 2.1 Artifact Metamodel

2.6.6 SACM Abstractions

The assurance case pattern specification is supported in SACM by properties defined in the base classes. All **SACMElement** has an *isAbstract* property, which means if the elements are abstract or non-abstract, and if the **SACMElement** is non-abstract it is possible to set its *abstractForm* property which is a reference to the abstract **SACMElement** from which it has been instantiated.

The **ModelElement** may have zero or more **ImplementationConstraints** indicate the conditions that must be fulfilled in order for instantiating a given **ModelElement**, (OMG, 2019). However the **ImplementationConstraint** is still too generic, i.e., there is no specific types as in the GSN. Thus, the SACM **IimplementationConstraint** can be defined as an element that gives multi-language textual information about how to instantiate a model element.

The **Terms** and **Expressions** allow an assurance case pattern to be created through the **ModelElement** *description* property. A Description has a *content* property, which is the type of **MultiLangString**, with it is possible adding multi-language description for the **ModelElement**. A Description element may contain **ExpressionLangStrings**, i.e., an element that references to **ExpressionElement**. An ExpressionElement provides the necessary placeholders to instantiate an assurance case pattern. The *name* property of a **ModelElement** can also be an **ExpressionLangString**, i.e., the *name* can also contain placeholders.

2.7 Automate Pattern Instantiation

This section describes the behaviour of a generic instantiation program. This section also discusses the representation and validation of **ImplementationConstraints** and the problem in building a generic instantiation program for the SACM version 2.1.

2.7.1 Instantiation Program

Model-based techniques can be used to automate assurance case pattern instantiation, i.e., enabling the automatic generation of an assurance case via instantiation of a pattern specification using information from system models. This approach has been earlier implemented in a tool that supports the instantiation of GSN assurance case patterns. Such approach is built upon a weaving model that provides mappings linking abstract elements of a GSN pattern, named **Role**, to elements from other XMI meta-models, e.g., system design. In such approach, the weaving model, the GSN pattern specification, and the required system models are inputs to an instantiation program, built using EOL. The instantiation program replaces the **Roles**, i.e., abstract *Terms*, of the pattern. This program replaces the abstract *terms* of the pattern with the information contained in the system models which have been mapped in the weaving model, (HAWKINS et al., 2015). However, constructing a model-based approach to automate the SACM pattern instantiation will be different from the GSN approach, because according to Wei et al. (2019), a abstract **Term** in the SACM, i.e., an abstract *term* of the pattern, should has an **ImpementationConstraint** that stores a 'query' in its *content* property. This 'query' gives the information to search in the system model defined in **Term** *externalReference* property for obtaining values for instantiating this abstract *term*. Therefore, there is no need for a weaving model to specify the mapping links between abstract *terms* and model elements since the queries stored into **ImplementationConstraints** can be used to automatically obtain the information from the system models. System models can be built upon different technologies, e.g., text and XMI files.

Therefore, the model-based approach allows the instantiation program to be created, and this program will be responsible for replacing all the abstract *terms* with the required information in order to instantiate the pattern. Due to the nature of the model-

based approach, the information for instantiating the abstract *term* can be obtained from any system model, and these system models can be developed in different technologies, e.g., a text requirement document and a XMI components diagram.

The construction of an instantiation program that is able to get the information no matter what technology the model has bee developed is hypothetical because nowadays there are many different technologies and with the advancing of the industry new technologies will arise. Therefore, to simplify the construction of an instantiation program, it is useful to reduce the set of model technologies, in favor of the ones in which the system models are developed. For example, if the required system models are stored into XMI files, an instantiation program should be created to search for information in XMI files. The instantiation program of Hawkins et al. (2015) is an example of this.

It is possible to imagine a generic instantiation program. This program should have extension points/hot spots that must be implemented to get required information in the required model, e.g., if the model is an XMI model, an extension is implemented to get the information in this type of model, so there is no need to create or understand the whole instantiation program, one need only understand how to get the model information and how to send it to the program. Thus, this generic instantiation program comes close to the ideal, hypothetical instantiation program, making it easier to get information in diverse system models specified in different technologies.

2.7.2 A Potential Instantiation Program for SACM 2.1

A potential program for instantiating SACM patterns should manage the instantiation of implementation constraints stated in the patterns. In the GSN, **optional**, **multiplicity** and **choice** constraints stated in a pattern have their own instantiation form and restrictions. For example, the GSN **choice** constraint can only be added to GSN supportedBy relationships. The GSN clearly establishes a distinction between **multiplicity**, **optional** and **choice** constraints 2.7. In GSN, each one of these constraints has a meaning and a graphical representation, e.g., a filled circle denotes **multiplicity** and a non-filled circle denotes **optional**, while a **choice** is represented by a filled diamond.

From the analysis of the SACM meta-model, it was not found a way of repre-

senting and establishing distinction among **ImplementationConstraints** in SACM as provided by GSN. The lack of distinction among constraints was a design decision of the OMG committee that could be a problem for the meta-model implementation in a domain-specific language and modeling tool. Therefore, for a safety engineer who needs to specify an assurance case pattern for automatic instantiation, it would be needed to find other ways to specify and representing different types of constraints in SACM. The suggestion of Wei et al. (2019) is that it is possible to use the *content* of an **ImplementationConstraint**, attaching in it a **LangString** element. This **LangString** can be used to store a string with the specification of an OCL constraint that describes how to get the required information from system models to automatic instantiation of an abstract term stated in an assurance case pattern. This constraint substitutes the necessity of a weaving model. However, the usage of OCL expressions for specifying constraints in SACM patterns impose issues that should be considered in the implementation of the instantiation program. The Epsilon Object Language can be used to interpret the content property of **ImplementationConstraint** elements specified in OCL, EOL, Structured Query Language (SQL) and other languages.

In the SACM 2.1 meta-model, the type of the *content* property of a **ImplementationConstraint** element is **MultilangString** type. A **MultilangString** provides multilanguage support, i.e., the internationalization of a content. The *value* attribute of **MultilangString** type allows the addition of various **LangString** elements. **LangString** is an element with a defined language and a string content. Therefore, for the construction of a generic instantiation program for SACM 2.1 assurance case patterns, one additional step is needed. This step is to know how to get the required information specified in different languages. However, it is not possible to construct a generic instantiation program that enables interpreting the required information stored into a **MultilangString** element in any existing language.

It is important to highlight that a specific **ImplementationConstraint** element, used to instantiate abstract *terms*, may also have constraints. For example, a given **ImplementationConstraint** element, equivalent to the GSN **choice** constraint, would be restricted to relations that involve a SACM **Claim**. However, the SACM 2.1

meta-model does not provide an explicit way to specify restrictions over **ImplementationConstraint** elements. An **ImplementationConstraint** element is too generic, and its unique restriction is that it can only be added to a **ModelElement**, but there is no limit in the number of **ImplementationConstraint** that can be added to a given **ModelElement**. Such flexibility offered by the SACM meta-model imposes challenges on automating the instantiation of SACM assurance case patterns.

The main challenges in the construction of a generic program for automating the instantiation of SACM assurance case patterns are: *i)* finding a standard way to embed the pattern instantiation information into an **ImplementationConstraint** *content* property for allowing the instantiation program get such information, whatever this information and its language are; *ii)* establishing a way to manage potential validation rules over the types of constraints and their possible instantiation information; *iii)* and how to represent different types of constraints simplifying the construction and understanding of an assurance case pattern.

3 SACM ACEditor Development Process

This chapter describes the process enacted by the author in the development of the SACM ACEditor tool. Section 3.1 provides an overview of the whole process and a description of each phase. Section 3.2 describes the activities related to the first phase. In Section 3.3 is presented the activities defined in phase 2. In Section 3.4 is detailed the activities related to phase 3. Finally, Section 3.5 explores the phase 4.

3.1 Overview

The SACM ACEditor has been developed integrated with EMF and GMF platforms using the Epsilon tools. The SACM ACEditor is built upon the Ecore meta-modeling language. The EMF meta-model editor can be used to create other tools for handling SACM 2.1 assurance cases. Therefore, it is possible to create newer plug-ins for validation, transformation, wizards, and other tools. Figure 3.1 shows an overview of the process followed by the author in the development of SACM ACEditor. The process comprises four phases: i) Specify the DSL Elements and their Graphical Representations; ii) EuGENia Automatic Generation; iii) Creating Complementary Plugins; iv) Fixing the Plugins. They are detailed in the following.

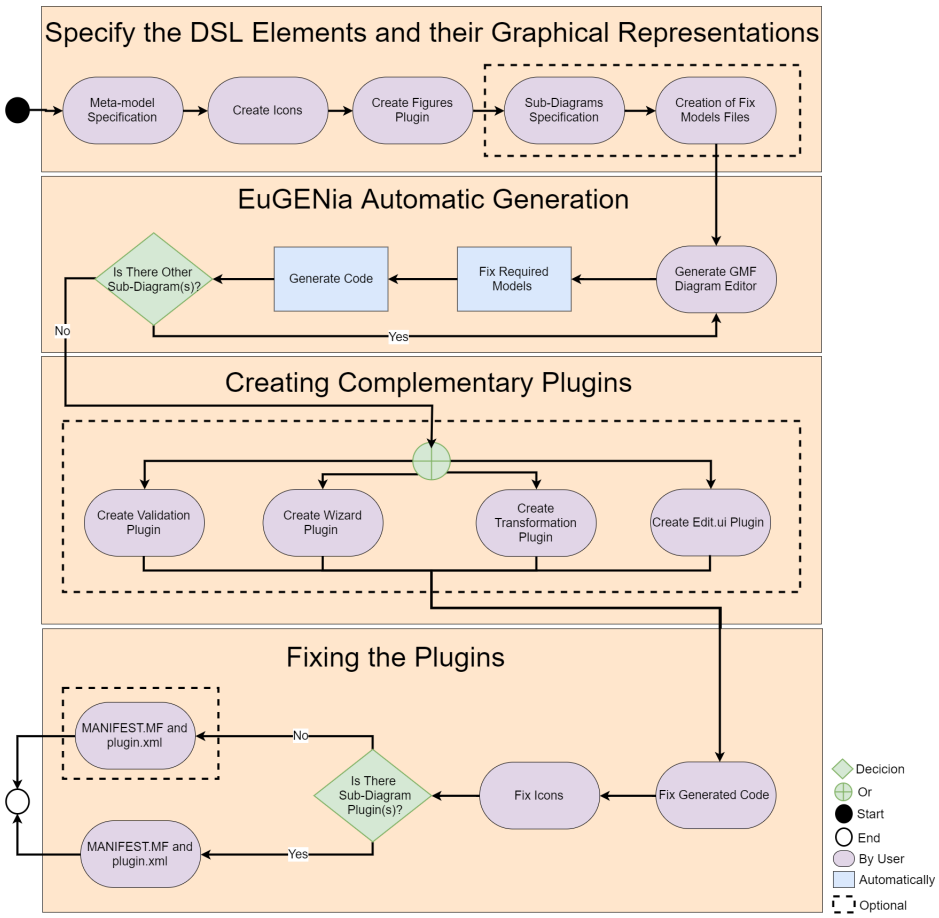

Figure 3.1: Development Process Overview

3.2 Specify the DSL Elements and their Graphical Representations

In this phase, the Emfatic file, which defines the meta-model elements and their mappings to graphical representations, is created. It is important to highlight that it is necessary to provide custom implementations for figures related to graphical representations of meta-model elements that are not present in the standard figures from GMF platform. After specified these figures using Emfatic annotations, i.e., the map linking meta-model elements to these figures, they should be implemented. For complex meta-models that contain multiples packages, such as the case of SACM meta-model version 2.1, it is recommended partitioning the specification of the meta-model into multiple Emfatic files. Each Emfatic contains the specification of model elements and their mappings to figure elements related to a package/portion of the meta-model, resulting in a set of Emfatic

files. Each Emfatic file contains the specification of a sub-model GMF editor plugin. Engineers can also create scripts in EOL to enable the EuGENia tool to fix automatically the meta-model before code generation.

3.2.1 Meta-model Specification

Input: the domain knowledge and the meta-model. **Purpose:** The main objective of this activity is creating an Emfatic file with a meta-model specification. This specification contains the declaration of classes and other types of elements of the meta-model, their attributes and relationships. Therefore, the meta-model specification is described in an .emf file using the Emfatic language. In order to build a GMF Diagram Editor, annotations are used for associating the meta-model elements to their graphical representations and icons. **Output:** the specification of a domain-specific language in a .efm file.

Performing this activity for developing the SACM Assurance Case Editor, yields the SACM meta-model specification in an Emfatic file and mappings linking SACM elements, e.g., **Claim** and **ImplementationConstraint**, to their respective graphical representations using the *gmf annotations* as illustrated in Listing 3.1. the GMF has been used to define which SACM elements are nodes and links. Due to limitations in representing SACM relationships as links within the GMF platform related to representing a link with multiple sources and targets, SACM ArtifactAssetRelationship and AssertedRelationship were defined stated as nodes instead links. Such strategy enables both specification and visualization of multi-source and multi-target SACM relationships.

In order to solve the problems discussed in Section 2.7.2, related to how differentiate **ImplementationConstraints**, to allow different representations and their own validation, the author have added a *type* property into the implementation constraint model element specification on the Emfatic file (line 3 from listing 3.1).

```
1 @gmf.node(...)
2 class ImplementationConstraint extends UtilityElement{
3       attr String [0..1] type;
4 }
```

Listing 3.1: **ImplementationConstriant** specification on Emfatic

Table 3.1 shows the SACM elements properties that on the meta-model specification have been mapped to default link styles of GMF. Thus, when these properties are defined, the respective links are shown in the editor's canvas.

Element	Property	Mapping
SACMElement	*abstractForm*	
SACMElement	*citedElement*	
AssuranceCasePackageInterface **ArgumentPackageInterface** **ArtifactPackageInterface** **TerminologyPackageInterface**	*implements*	
AssuranceCasePackageBinding **ArgumentCasePackageBinding** **ArtifactCasePackageBinding** **TerminologyCasePackageBinding**	*participantPackage*	
ArtifactReference	*referencedArtifactElement*	
Term	*origin*	
ArgumentReasoning	*structure*	
AssertedRelationship	*reasoning*	
Assertion	*metaClaim*	

Table 3.1: Custom Links of SACM ACEditor

3.2.2 Create Icons

Input: the meta-model specification in Emfatic file, focusing on the required declared icons. **Purpose:** In this activity, the engineer creates the icons for the GMF Diagram Editor. For each required icon that is stated in the Emfatic file, an image should be created. **Output:** all images required for the pallet of a GFM Diagram Editor of the targeted language are created. Figure 3.2 illustrates the icons that have been defined for the SACM ACEditor.

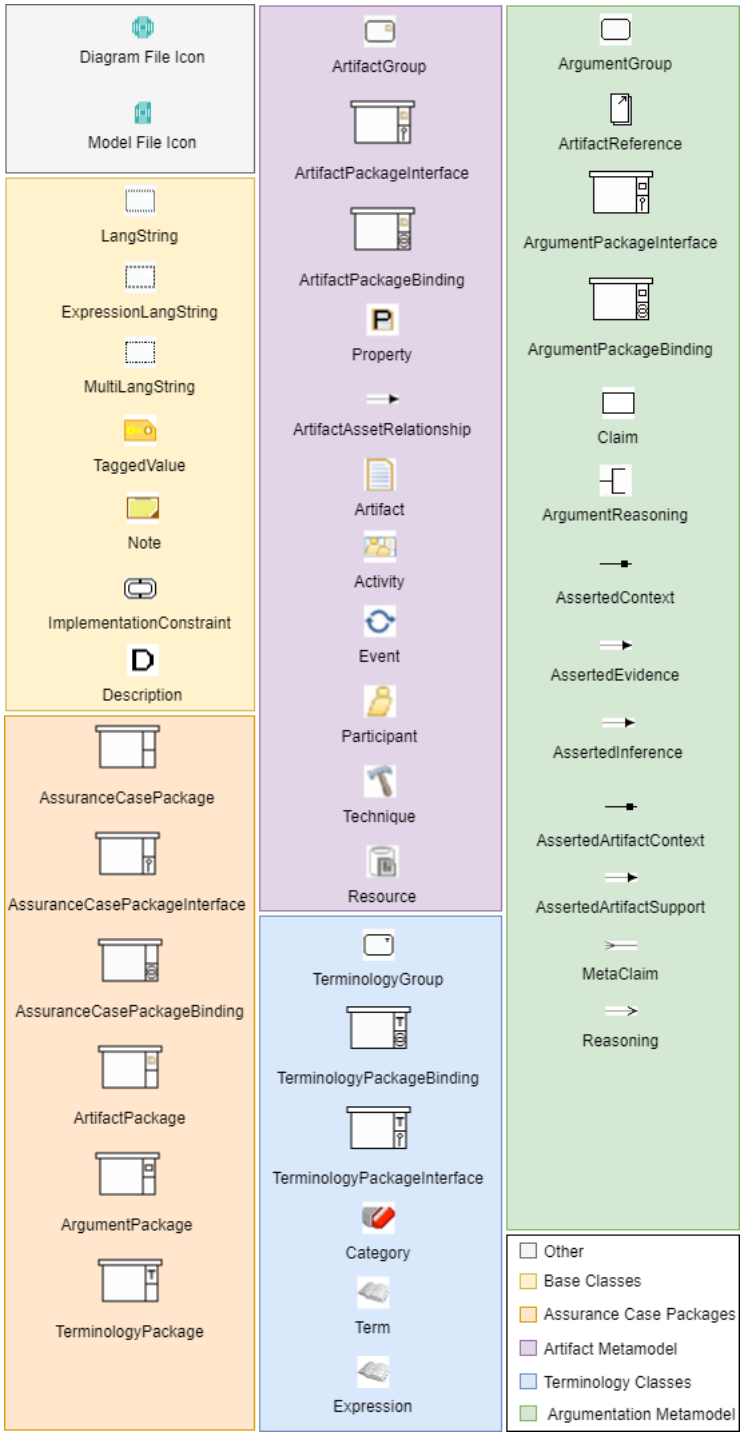

Figure 3.2: Icons of the SACM ACEditor

3.2.3 Create the Figure Plugin

Input: the meta-model specification in an Emfatic file, focusing on the required custom figures that need to be implemented. **Purpose:** In this activity, the engineer creates a new Eclipse plugin project containing the implementation of figures that will be drawn on the canvas of a GMF Diagram Editor. Each figure must be implemented as a Java class

that inherits directly or indirectly from "org.eclipse.draw2d.Figure". The plugin project where the Emfatic file is stored has a dependence relationship with this Figure Plugin. Later, the created figure classes are referenced in the path of *gmf annotations* associated with each model element in the Emfatic file. **Output:** a new Plugin project with the implementation of all required custom figures. The figure classes are further used by the GMF Diagram Editor to show the elements in the tooling pallet and drawing the graphical elements in the canvas.

In order to simplify the view configuration on the SACM ACEditor, the hierarchical structure of the meta-model has been used in declarations of figure elements. Since most elements of SACM have support multi-language , in order to adapt this support the visualization of it has been limited to one language in the element figures. The SACM ACEditor allows the user to change the language, updating automatically all the figures. Figure 3.3 shows the figure implementations that are different from their icons. The other figures are the same as their icons.

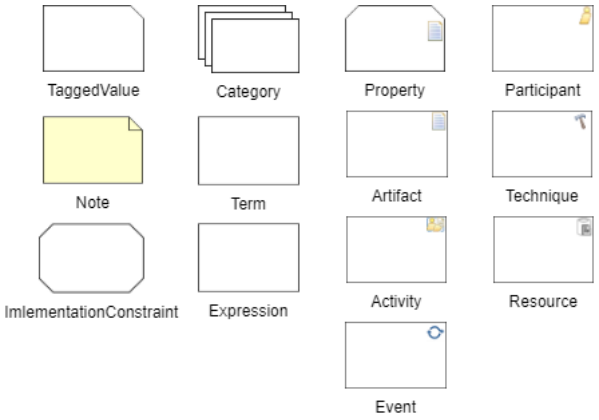

Figure 3.3: Implemented Figures Different from SACM ACEditor Icons

3.2.4 Sub-Diagrams Specification

Input: the Emfatic file of meta-model specification. **Purpose:** In this activity the engineer must decompose the input Emfatic file into other Emfatic files contained sub-diagrams specifications. Therefore, if there is no sub-diagram, this activity is not performed. This activity must be executed if it is needed navigating among different GMF Diagram Editors, i.e., via double click on an element that represents a sub-diagram, a

new canvas is opened showing its content.

An Emfatic file can be used to generate the GMF Diagram Editor for the meta-model. The first class stated under `@gmf.diagram` annotation in an Emfatic file must be a non-abstract class. This class defines the possible elements that can be displayed in the canvas of a GMF Diagram Editor and it is the root class of this diagram. It is simply creating a sub-diagram from an Emfatic input file. Basically, the sub-diagrams are copies from this input Emfatic file, switching only the class under the `@gmf.diagram` annotation. These sub-diagrams have all the elements specified on the original Emfatic file. However, in order to reduce the number of elements of the SACM GMF Editor pallet, the annotations of the elements that do need to be displayed in sub-diagrams should be removed.

The properties *uri* and *package* of all sub-diagram specifications are the same as the original Emfatic file. Therefore, the EMF Editor, Edit and Test plugins, and the model code are the same for each sub-model editor. Only the GMF Diagram Editor is different. **Output:** a set of Emfatic files. In the development of GMF Diagram Editor for the SACM meta-model, it was created Emfatic files for: **AssuranceCasePackage**, **ArtifactPackage**, **ArgumentPackage**, **TerminologyPackage** and **ModelElement**.

3.2.5 Creation of Fix Model Files

Input: zero or more Emfatic files. **Purpose:** In this activity, the engineer creates the files, i.e., Ecore2GMF.eol, FixGMFGen.eol and Ecore2GenModel.eol, for fixing the generated models, required for code generation. This is obligatory if the models required to generate the editors need some fixes/customization. This activity may change according to the input models. If there are no sub-diagrams in the editor, i.e., there is only one Emfatic file with a single model specification, this activity is optional. On the other hand, if there are sub-diagrams in the editor, this activity should be performed for linking the sub-diagrams. These links are needed to redirect to a given sub-diagram when the user gives a double click in a container element in the editor's canvas. Part of this activity is an adaptation of tutorial purposed by Wright (2005). This activity is focused on partitioning a GMF Diagram Editor. **Output:** Ecore2GMF.eol, FixGMFGen.eol

and Ecore2GenModel.eol files. These files are further used by EuGENia to automate fixes and changes in the required models during the code generation for the editor. The available models for changing are the Ecore, GMFGraph, GMFTool, GMFMap, GMF GenMode, EMF GenModel. Section 2.2.3 explains the EuGENia code generation process, the aforementioned models and the required files to change these models.

Output: the EOL files that automates fixes in models files, i.e., Ecore2GMF.eol, FixGMFGen.eol and Ecore2GenModel.eol. The output can be one or more of them, it depends which models need fixes/changes for editor generation. For example, to automate the addition of Figures Plugin dependence into GMF Diagram Editor, the FixGMFGen.eol shuld be created with the pseudo-code bellow:

```
GmfGen!GenPlugin.all.first().requiredPlugins.add(<Figures Plugin ID>);
```

If there are sub-diagrams, the minimum fixes should be specified in FixGMF-Gen.eol and in ECore2GMF.eol files. The ECore2GMF.eol file changes the GMFMap model in order to allow the redirection via double-clicking on a container element. The FixGMFGen.eol changes the GMF GenModel model linking the respective GMF Diagram Editor that must be opened when a redirecting action is triggered by the user. The code fixes on the ECore2GMF.eol defines the container elements that need redirection. Therefore, if an element needs redirection, its **Node Mapping** on the GMFMap model needs a related diagram on *RelatedDiagrams* property. This related diagram must be a **Canvas Mapping** type as illustrated in Listing 3.2. The property *DiagramNode.name* of a **Node Mapping** (line 2) has been used to verify if there is a need for redirection.

```
1 var nodesMapping = GmfMap!NodeMapping.all;
2 if(nodesMapping needs redirection){
3     var canvas GmfMap!CanvasMapping.all.first();
4     nodesMapping[i].RelatedDiagrams.add(canvas);
5 }
```

Listing 3.2: Minimum Fixes on GMFMap

The fixes on the FixGMFGen.eol depend on the minimum fixes written in the file ECore2GMF.eol. Therefore, it is not possible to configure a redirection if the element

does not allows it. Therefore, the fixes on the FixGMFGen.eol file complements the fixes written in ECore2GMF.eol, allowing the correct redirection to sub-diagrams.

- Fist step, it is necessary to ensure that each GMF Diagram Editor is unique, thus:

  Make the *Domain File Extension* property of **Gen Editor Generator** unique;

  Make the *ModelID* property of **Gen Editor Generator** unique;

  Make the *Package Name Prefix* property of **Gen Editor Generator** unique;

  Make the *ID* property of **Gen Plugin** unique;

  Make the *Name* property of **Gen Plugin** unique;

- Second step, using the values provided by the step above, it is necessary to link the sub-diagrams in order to open the respective GMF Diagram Editor. For each **Open Diagram Behaviour** that needs redirection, change their properties as bellow, otherwise, i.e., if it do not redirect, delete it:

  Change *Diagram Kind* property to respective *ModelID*;

  Change *Editor ID* property to respective *ID*;

  Make *Edit Policy Class Name* property unique for each sub diagram redirection.

The Listing 3.3 shows a generic example of how to make the fixes above in a FixGMFGen.eol file. In the development of the SACM ACEditor, it has been used the property *Context* of **Gen Context Menu** for obtaining the diagram class name. The *Subject.editPartClassName* property from **Open Diagram Behaviour** has also been used to filter the classes that needs redirection.

```
1 var genEditorGenerator =  GmfGen!GenEditorGenerator.all.first();
2 var genPlugin = GmfGen!GenPlugin.all.first();
3 genEditorGenerator.DomainFileExtension=<Respective Domain File Extension>;
4 genEditorGenerator.ModelID=<Respective Model ID>;
5 genEditorGenerator.PackageNamePrefix=<Respective Package Name Prefix>;
6 genPlugin.ID=<Respective GMF Editor Plugin ID>;
7 genPlugin.Name=<Respective  GMF Editor Plugin Name>;
8 for (odb in GmfGen!OpenDiagramBehaviour.all){
9     if (odb needs redirection) {
```

```
10          odb.DiagramKind= <Respective Kind>;
11          odb.EditorID= <Respective GMF Editor Plugin ID>;
12          odb.EditPolicyClassName= <Respective Edit Policy Class Name>;
13      }
14  }
```

Listing 3.3: Minimum Fixes on GMF GenModel

It is important to highlight that if there is an annotation `@gmf.compartment` on the Emfiatic specification in all elements that need redirection, the fixes written in the ECore2GMF.eol are not needed. By default, in the EuGENia, a compartment element allows the redirection to another diagram. Therefore, for the fixes in FixGMFGen.eol file, if the element which has the compartment annotation should not redirect to another diagram, its **Open Diagram Behaviour** must be deleted.

3.3 EuGENia Automatic Generation

In this phase, for each Emfatic file, the EuGENia will automatically generate the implementation code. The way to start this phase is by right click on the diagram/sub-diagram file, i.e., its Emfatic specification, and selecting, in the context menu, the option 'EuGENia > Generate GMF editor'. The activities of this phase are automatically executed. However, It is necessary to start this phase on each diagram/sub-diagram. Thus, the generation of GMF editor should be executed multiples times if there is more than one Emfatic file. Section 2.2.3 describes the workflow of automatic editors generation done by EuGENia. The result of this phase, for each Emfatic specification, yields: the GMF Diagram Editor(s) Plugin(s), EMF Editor Plugin, EMF Edit Plugin, EMF Test Plugin, and Model Code. At this point, the Editor should have been functional, i.e., it is possible to create and edit models. However, some fixes are necessary in order to link model changes with possible view changes and other general properties.

3.3.1 Generate GMF Diagram Editor

Input: a diagram/sub-diagram file. **Purpose:** the purpose of this activity is to create the Annotated Ecore Metamodel, EMF GenModel, GMF GenModel, GMFTool, GMF-

Graph and GMFMap models required to generate both EMF and GMF editor's code. This activity is automated by EuGENia, EMF and GMF platforms. **Output:** the annotated Ecore Metamodel, EMF GenModel, GMF GenModel, GMFTool, GMFGraph and GMFMap. On SACM ACEditor, each execution of this activity by the execution of this phase has generated these outputs models for one sub-diagram. This phase have been started firstly with **AssuranceCasePackage** sub-diagram, secondly with **ArtifactPackage** sub-diagram, thirdly with **TerminologyPackage** sub-diagram, fourthly with **ArgumentPackage** sub-diagram, and lastly with **ModelElement** sub-diagram.

3.3.2 Fix Required Models

Inputs: the Annotated Ecore Metamodel, EMF GenModel, GMF GenModel, GMFTool, GMFGraph, GMFMap, Ecore2GMF.eol, FixGMFGen.eol and Ecore2GenModel.eol. **Purpose:** the purpose of this activity is execute the EOL code specified in the Ecore2GMF.eol, FixGMFGen.eol and Ecore2GenModel.eol in order to polish/fix/change the The Annotated Ecore Metamodel, EMF GenModel, GMF GenModel, GMFTool, GMFGraph, GMFMap models. This activity is automated by EuGENia via execution of the code specified in the Ecore2GMF.eol, FixGMFGen.eol and Ecore2GenModel.eol files. Therefore, the input models are fixed/polished in this process. **Outputs:** the Polished Annotated Ecore Metamodel, EMF GenModel, GMF GenModel, GMFTool, GMFGraph and GMFMap.

The repeated execution of this phase have been the polished the models for each sub-diagrams, i.e., the models for **AssuranceCasePackage**, **ArtifactPackage**, **TerminologyPackage**, **ArgumentPackage**, **ModelElement**.

3.3.3 Code Generation

Input: the Polished Annotated Ecore Metamodel, EMF GenModel, GMF GenModel, GMFTool, GMFGraph, and GMFMap. **Purpose:** the main purpose is generating the GMF Diagram Editor Plugin. However, due to its dependence on the EMF Editor, if the EMF Editor, Edit and Test plugins and the model code have not been generated, they are generated before the generation of the GMF Diagram Editor Plugin. This activity is also

automated by EuGENia, which generates the GMF Diagram Editor Plugin. **Output:** the GMF Diagram Editor Plugin, EMF Editor Plugin, EMF Edit Plugin, EMF Test Plugin, and Model Code.

As the final result of the execution of this phase on the development of the SACM ACEditor, the EMF Editor Plugin, EMF Edit Plugin, EMF Test Plugin, Model Code and a set of GMF Diagram Editor Plugins have been generated. The set of GMF Diagram Editor Plugins consist in GMF Editors for the following sub-diagrams: **AssuranceCasePackage**; **ArtifactPackage**; **TerminologyPackage**; **ArgumentPackage**; **Model Element**.

3.4 Create Complementary Plugins

3.4.1 Create Validation Plugin

Input: the meta-model specification. **Purpose:** creating a new Eclipse Plugin project responsible for all validations required by the meta-model not provided by the EMF core. In this phase, the complementary plugins are developed. All the activities of this phase are optional and there is not order to execute them.

The core of the EMF model already does a few validations. However, if there exist constraints in the meta-model indicating that the model is not valid, a validation plugin is required. This plugin will be responsible for the model validation, i.e., it verifies if a model conforms to all meta-model constraints. **Output:** a validation Plugin.

The developed validation Plugin for the SACM ACEditor comprises an EVL file with the validation rules defined in the meta-model. This plugin implements the `org.eclipse.epsilon.evl.emf.validation` extension point, with a new **constraintBinding**, its *namespaceURI* field with the value of the model *nsUri* defined in the Emfatic file, and the *constraints* field with the path of the EVL file (EPSILON, c).

3.4.2 Create Wizard Plugin

Input: the meta-model specification. **Purpose:** creating a Plugin to provide customized capabilities for the users that can or not modify the actual model. A wizard plugin allows

users creating capabilities available to the users in a context menu. The users can access these capabilities via right-click in the canvas or in an element and selecting the *wizard* option. A wizard plugin can perform in-place modifications in a model. **Output:** a Wizard Plugin.

In the development of the SACM ACEditor, a wizard plugin has been developed as an implementation of an Epsilon extension point. The extension point for the wizard plugin was specified in an EWL file that contains the capability/options to be available to the users. Thus, the `org.epsilon.ewl.eclipse.gmf.wizards` extension point has been implemented to integrate the developed wizard to the SACM GMF Diagram Editor (KOLOVOS et al., 2007). From this extension must, it was created a new **wizard**, with *namespaceURI* field, which is the value for the model *nsUri* defined in the Emfatic file, and the *file* field with the path of the EWL file.

3.4.3 Create a Transformation Plugin

Input: the meta-model specification. **Purpose:** creating a transformation plugin responsible for executing model to model transformations. A transformation plugin is created when it is needed to convert a source EMF model into an equivalent target model. However, the model transformation plugin is not integrated within GMF as the wizard and validation plugins. The transformation can be specified as a program using Epsilon Transformation Language (ETL). The user can perform a model transformation via the execution of an EOL program specified in ETL. The model to model transformation should be developed as an independent plugin not related to the GMF Diagram Editor Epsilon (b). A model transformation plugin receives an input source model and transforms it into an equivalent output model in the target language. **Output:** a model to model transformation plugin.

In the development of the SACM ACEditor, a stand-alone approach has been adopted to create a transformation plugin. The developed plugin has a class that implements the `org.eclipse.popupMenu` extension point, responsible for initiating the standalone model transformation. When the user gives a right-click in a source model on the Eclipse Navigation bar, a popup menu appears with a menu item to transform the model.

When this option is selected GSN model is transformed into an equivalent SACM 2.1 model. The SACM ACEditor only supports the transformation from GSN to SACM 2.1 assurance case models. The extension of the GSN model supported by the transformation plugin is '.gsmetamodel'.

3.4.4 Create Edit.ui Plugin

Input: the meta-model implementation. **Purpose:** creating a new Plugin project in order to group all the views that interact with the user, and providing custom views to change the element properties in the EMF "Property View". An Edit.ui plugin is responsible for the editor's **PropertySource**, i.e., what will be displayed to edit the value of a property of a selected element on the Eclipse "Property View". For example, if a property "name" from an element "E" needs to open a custom dialog to change its value. **Output:** a Edit.ui Plugin.

The **CustomPropertySource** is the editor's custom **PropertySource** , used by both EMF and GMF Diagram editors. In order to do this, the following steps should be performed:

- Step 1: create the `CustomPropertySource` by:

  Extending `org.eclipse.emf.edit.ui.provider.PropertySource`; and

  Overriding the `createPropertyDescriptor(IItemPropertyDescriptor ipd)`.

- Step 2: create custom dialogues or non-default dialogues to change properties value.

  The method `createPropertyDescriptor(...)` of a **CustomPropertySource** should return the respective **PropertyDescriptor** of the selected property. Therefore, for each property that needs a custom edit value view, a class that extends **PropertyDescriptor** should be created. This class should override the method `createPropertyEditor(...)` that returns what will be displayed to edit a property value.

- Step 3: linking the EMF Editor with the **CustomPropertySource**:

  In the MANIFEST.MF file, the developer should add the Edit.ui plugin on the EMF Editor dependencies.

In **editor's** presentation package, the `getPropertySheetPage()` method, which creates a new **PropertySheetPage**, should set its **PropertySourceProvider** to a new instance of **AdapterFactoryContentProvider**, and overriding the method `createPropertySource(...)`. This is needed to return a new instance of the class **CustomPropertySource**.

- Step 4: linking the GMF Diagram Editor to the **CustomPropertySource**:

  In the MANIFEST.MF file, the developer should add the Edit.ui plugin on EMF Editor dependencies.

  All **PropertySection** into the 'sheet' package of GMF Diagram Editors should return a new instance of **CustomPropertySource** on `getPropertySource()` method, which also needs to be overriden.

The '**EPackage.eINSTANCE**' of the current model interface should be used to compare the `createPropertyDescriptor` **IItemPropertyDescriptor** parameter and the properties of the classes, i.e., to compare which property have been selected with the existing properties of all model classes. On the SACM ACEditor, this has been used to create custom views to edit the values of *externalReference* of a **Term**, the properties of type Date, e.g., **Activity** *startTime* and *endTime*, the property *element* of an **Expression**, the *lang* of **LangString**, the *lang* of **ExpressionLangString**, the *categories* of **ExpressionElement**, the *AbstractForm* of **SACMElement**, and the *type* of **ImplementationConstraint**. On the *type* of **ImplementationConstraint** property, this approach has been adopted to block the user changing the values.

3.5 Fix the Plugins

In this phase the generated code, icons and few meta-data of the Plugins should be fixed, thus, polishing the plugins generated by EMF and GMF platforms.

3.5.1 Fix Generated Code

Input: the GMF Diagram Editor and the EMF Editor plugins. **Purpose:** fixing the generated code to solve errors or link the changes in properties values with view changes. In this activity, the engineer fixes the generated code. This activity is optional depending on what the editor requires. On the other hand, in case of changes in element properties implying in changes in the element view and errors in the code, this activity is mandatory. **Output:** the GMF Diagram Editor and EMF Editor plugins without errors in their codes and the view elements linked with changes in their properties.

Since the SACM ACEditor have different sub-diagrams, e.g., **AssuranceCasePackage** and **ArtifactPackage**, some class of elements that one sub-diagram contain other can do it. For example, the **ArgumentPackage** sub-diagram editor contains **Claims** and the **ArtifactPackage** sub-diagram also contain **Claims**. Thus for some class of elements, the code will be repeated in multiple GMF Diagram Editors plugins. However, the link between changes in elements properties values and view changes will be the same for each element class. Therefore, the code that configures the element view, based on its class properties, can be the same for each class.

The abstract classes responsible for configuring the element view have been created for the most element classes. These classes extend the GMF **ShapeNodeEditPart** for nodes and **ConnectionNodeEditPart** for links. These classes should override the method `handleNotificationEvent(Notification notification)` that receives the element modifications via notification. Thus, when notified, the method `getPrimaryShape()` should be used to obtain the figure element and change it. Therefore, the classes in the GMF Diagram Editor edit.parts package should extend from these generic classes responsible to link the element properties changes with view changes. Also, the view should be configured when the view figure is created, allowing the saved changes on the model to be displayed as soon as the element is created. The methods `createNodeShape()` for nodes and `createConnectionFigure()` for links in the classes into the edit.parts package were also modified. These methods call the method responsible for viewing the changes in these generic classes. For example, `configureView(IFigure figure)` for links and `configureView()` for nodes.

There is a common error when the diagram domain is bigger. The Eclipse throws the exception "exceeding the 65535 bytes limit InternalDSLLexer.java". It occurs in the **NavigatorContentProvider** class into the GMF Diagram Editor navigator package. This class implements the **ICommonContentProvider** interface and has too many lines of code in the method `getViewChildren(View view, Object parentElement)`. Due the nature of this function, it contain a `switch` with a large number of cases. A simple way to fix this error is to create a new function for each case of this `switch`. In order to make the execution of these code fixes simpler and faster, a python code has been created. This code reads all archives of the workspace making the classes in the GMF Diagram Editor edit.parts package extending generic **EditPart** classes, and modifying the creation method from this class. The python code also fixes the possible error in the **NavigatorContentProvider**, breaking the `switch` into various methods.

3.5.2 Fix Icons

Input: the GMF Diagram Editor, EMF Editor, and EMF Edit plugins. **Purpose:** replacing the default icons that have been automatically generated into the *icons* folder from the editors by custom icons, i.e., the editor's icons. In this activity, the engineer fixes the icons from EMF and GMF editors. **Output:** the GMF Diagram Editor, EMF Editor and Edit plugins with custom icons.

In the development of the SACM ACEditor, the icons that require replacement are kept into the *icons* folders from the plugins. Therefore, after the generation process, an external folder has been created containing all editor's custom icons. For the creation of these custom icons folder, the icons into each *icons* folder has been copied and modified. This approach allowed the automation of this activity after its first execution, i.e., the creation of a custom icons folder. It has been created a python program that replaces the icons stored into the *icons* folder of each input plugin to their respective icons in the custom folder. This approach is interesting because if it is needed to recreate all the plugins due to few modifications on the Emfatic files, there will be no need to change all these icons manually. However, if the modifications into the Emfatic file change the icons, the custom icons folder needs to be revised.

3.5.3 MANIFEST.MF and plugin.xml

Input: the GMF Diagram Editor Plugins. **Purpose:** fixing the meta-data of the input plugins by defining which diagram(s) will be the root(s) in the Editor. A GMF Diagram Editor Plugin can represent a diagram or sub-diagram, thus it is necessary to define which will be the root(s) diagram(s). **Output:** the GMF Diagram Editor Plugins with the correct meta-data configuration.

In the SACM ACEditor, the root class of the diagram is the **Assurance Case Package**. In order to limit it, the `org.eclipse.ui.newWizards` extension point have been removed from the GMF Diagram Editors plugins that represent the **Artifact Package**, **Terminology Package**, **Argument Package** and **Model Element**. Therefore, the user can not create a diagram file with these elements as root. This extension point has been removed from the EMF Editor plugin.

The model creation name and description of the **Assurance Case Package** GMF Diagram Editor Plugin have been changed via modification on the 'newWizardName' and 'newWizardDesc' in the "plugin.properties" file. The name for the diagram file creation has been defined as 'Assurance Case Diagram (SACM 2.1)', and its description to create a new Assurance Case diagram according to Structured Assurance Case Metamodel version 2.1 (SACM 2.1).

4 SACM ACEditor Architecture

This chapter provides an overview of the components from the SACM ACEditor and a brief exploration of each one. Section 4.1 provides an overview of the SACM ACEditor architecture. In Section 4.2 is provided a brief explanation of SACM editor components. Finally, Section 4.3 provides the motivation and explanation about the SACM editor extension points.

4.1 Overview

This diagram shows the relationships between the developed components. The 'Use' relationship occurs when a component has a dependence on others, i.e., it requires the other component to execute. The 'Extension' relationships are when a component has its extension point implemented by another component. Figure 4.1 describes the architecture of the SACM ACEditor, which has been developed on Eclipse platform, thus it consists of a set of Eclipse Plugins. The plugins with similar characteristics have been grouped in packages in the figure to make easier the understand and to show their common relationships with other plugins.

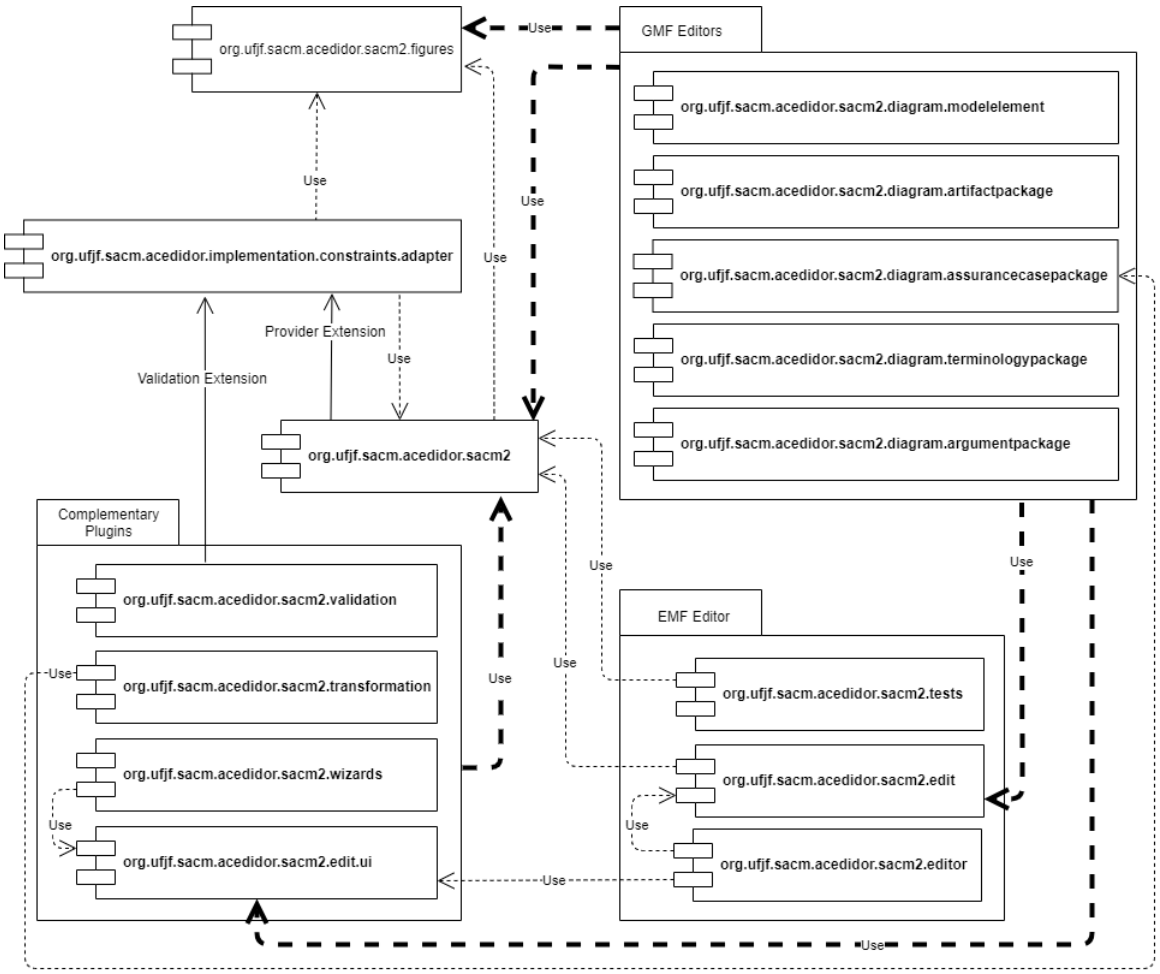

Figure 4.1: Components Diagram of SACM ACEditor

The plugins into the GFM Editors package consist of the set of GMF Diagrams Editors for each sub-diagram of the domain model, i.e., SACM 2.1 metamodel. Into the EMF Editor package, there are the plugins generated by the build of the EMF Editor. The Complementary Plugins package contains the plugins which are not a mandatory requirement to built the Editor, they have been created in order to add functionalities to the Editor such as validation and transformation or to improve the interaction with the user, i.e. wizards and the edit.ui plugin.

Due the addition of a property *type* into **ImplementationConstraint** 2(two) extension points have been developed and implemented by a component. These extensions are explored with more details than the other components.

4.2 Components SACM ACEditor

This section briefly explains each component of the SACM ACEditor, and their relationships.

4.2.1 Figures Plugin

The `org.ufjf.sacm.aceditor.sacm2.figures` contains the java implementation of all figures used on canvas of GMF Diagram editors. It not require none of the other developed plugins.

4.2.2 Model Plugin

The `org.ufjf.sacm.aceditor.sacm2` contains the model implementation and also the Emfatics specifications of the domain models, i.e., the sub-diagrams models. This plugin requires the figures plugin because it has an extension point that uses the figures plugin's classes. This plugin contains also the generic edit parts class, i.e., the classes which the edit part classes in the diagrams plugins inherit, they responsible to link the view with the model and reduce the code repetition.

The 'Provider Extension' relationship with the adapter plugin is due an implementation of these extension by the adapter. Due the property *type* added in **ImplementationConstraint**, an implementation of this extension can provide its types of **ImplementationConstraint**.

4.2.3 GMF Editors Plugins

This package contains the GFM diagram editors for each SACM sub-model. They have been developed from the sub-diagrams of **ModelElement**, **ArtifactPackage**, **AssuranceCasePackage**, **TerminologyPackage** and **ArgumentPackage**.

- `org.ufjf.sacm.aceditor.sacm2.diagram.modelelement`: the GMF Diagram Editor for **Model Element** sub-diagram, with the addition of **Property** of **ArtifactAsset** by fixes. Thus, for any element which is not a **ArtifactPackage**, **AssuranceCasePackage**, **TerminologyPackage** and **ArgumentPackage** this diagram

editor is used to access their sub-diagrams;

- `org.ufjf.sacm.aceditor.sacm2.diagram.artifactpackage`: the GMF Diagram Editor for **ArtifactPackage** sub-diagram;
- `org.ufjf.sacm.aceditor.sacm2.diagram.assurancecasepackage`: the GMF Diagram Editor for **AssuranceCasePackage** sub-diagram, it is also the root diagram editor;
- `org.ufjf.sacm.aceditor.sacm2.diagram.terminologypackage`: the GMF Diagram Editor for **TerminologyPackage** sub-diagram;
- `org.ufjf.sacm.aceditor.sacm2.diagram.argumentpackage`: the GMF Diagram Editor for **ArgumentPackage** sub-diagram.

The GMF Editors Plugins require the respective edit plugin because it contains providers to display the model elements in a user interface. The relationship between these plugins and the model plugin is due to their edit parts classes extend from the generic edit parts classes. The relationship with the figures plugin is because the developed java figures are used in their edit parts classes, which are responsible to display the elements' figures on the GMF Diagram Editor canvas. Finally, the relationship with edit.ui plugin because for some elements custom dialogues have been developed to edit the value of a selected element property on 'Property View' of Eclipse, more details of it can be found in Section 3.4.4.

4.2.4 EMF Editor Plugins

The plugins into this package are related to the creation of the EMF Editor. They have been generated automatically, the tests and the edit plugin have not been modified.

- `org.ufjf.sacm.aceditor.sacm2.tests`: contains templates to write tests for the model implementation. It requires the model implementation code;
- `org.ufjf.sacm.aceditor.sacm2.edit`: the edit plugin contains Providers to display the model elements in a user interface, e.g., it offers labels for the model ele-

ments which can be used to display a model element showing an icon and a name. It requires the model implementation code;

- `org.ufjf.sacm.aceditor.sacm2.editor`: the editor plugin is used to create and modify instances of a model. It depends to edit plugin because it contains Providers to display the model elements in a user interface. The relationship with edit.ui plugin is necessary because for some elements custom dialogues have been developed to edit the value of a selected element property on 'Property View' of Eclipse, more details of it can be found in Section 3.4.4.

4.2.5 Complementary Plugins

In this package are the plugins which are not a mandatory requirement in order to built the Editor. However, they have been created in order to add functionalities to the Editor. These functionalities are a custom validation of the model with the constraints of SACM 2.1 metamodel, and a transformation of GSN models to SACM 2.1 models. And also to improve the interaction with the user. Although section 3.4 contains a detailed explanation of them, bellow there is a brief explanation of each of them.

- `org.ufjf.sacm.aceditor.sacm2.validation`: the 'Validation Extension' relationship is due the property *type* added in **ImplementationConstraint**. Therefore, an implementation of this extension can provide a validation of a **ImplementationConstraint** with defined type. It requires the model implementation code in order to send the elements to validate in 'Validation Extension';

- `org.ufjf.sacm.aceditor.sacm2.transformation`: this plugin does the transformation of GSN models to SACM 2.1 models. It require the root GMF Diagram Editor, i.e., `org.ufjf.sacm.aceditor.sacm2.diagram.assurancecasepackage`, in order to create the diagram file after execute the transformation;

- `org.ufjf.sacm.aceditor.sacm2.wizards`: it provides functionalities/options that can be accessed by right click in a element selecting 'Wizard' on context menu, improving the communication with users. It requires some dialogs of edit.ui plugin;

- `org.ufjf.sacm.aceditor.sacm2.edit.ui`: it contains a set of dialogues and the SACM ACEditor custom '**PropertySource**'.

All of these plugins require the model plugin. However, the transformation plugin requires the model plugins to get the available languages to transform the model, different from others which requires the model implementation code.

4.2.6 Adapter Plugin

The `org.ufjf.sacm.aceditor.sacm2.implementation.constraints.adapter` is the adapter plugin. It implements the 'Provider Extension' and 'Validation Extension'. Thus, it provides types of **ImplementationConstraints** and also their own validation rules. The types provided by this plugin are some of the GSN abstractions (**multiplicity**, **optional** and **choice**), in Section 2.5.1 there is a detailed explanation about them.

The 'Provider Extension' and 'Validation Extension' can have various implementations, following the structure of this adapter. Thus, it is possible provider various types of **ImplementationConstraints** an also different validations. In the next Section, there is a detailed explanation about this adapter and these extension points.

4.3 Extensions of SACM ACEditor

This section aims to describe the 2(two) extensions of the SACM ACEditor, i.e., 'Provider Extension' and 'Validation Extension' showed in the overview of the editor architecture in section 4.1. The 'Provider Extension' is an extension for providing types of **ImplementationConstraints** and the 'Validation Extension' provides a way to validate these types. It contains an overview of how the adapter plugin implements these extensions. Although they can be implemented separately, i.e., one plugin implements the types provider and another implements the validation for these types, they are not independent because a type is required for the validation.

4.3.1 Overview

Figure 4.2 shows the implementation of the extension points by the adapter plugin, it may exist different implementations of the validation and the types provider extensions. However, they can not have an intersection of types, i.e., they must not provide the same type(s) and they

must not valid the same type(s).

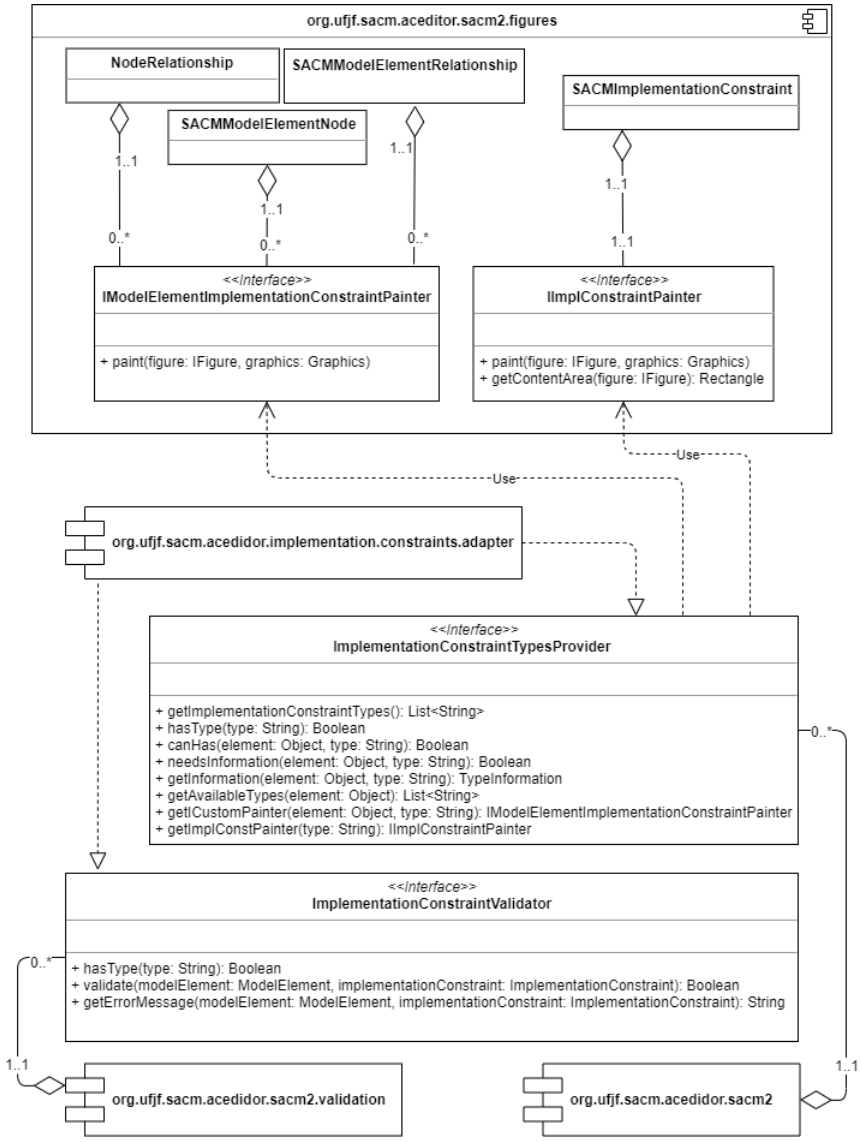

Figure 4.2: Extensions of SACM ACEditor

It is important to highlight the fact that with the addition of a property *type* on **ImplementationConstraints**, and the usage of the Eclipse extension points for providing the types and their validation rules gave more flexibility to **ImplementationConstraints**.

4.3.2 Provider Extension

The `org.ufjf.sacm.aceditor.sacm2.ImplementationConstraintTypesProvider` is the id of this extension point which an implantation of this extension point needs to use. From this extension must be create a new **Provider** whose the *Provider* field is the class which implements the **ImplementationConstraintTypesProvider** interface.

The implementations of this extension point can provide different types to be added in the model elements. These **ImplementationConstraints** with types can be added in the elements by the wizard plugin, it can be accessed by right-clicking in an element selecting option 'ADD ImplementationConstraint with a type' on the option 'Wizard' on the context menu, then a window is opened showing the available types for the selected element. Some problems showed in Section 2.7.2 which discuss an instantiation program for the SACM have been solved by this extension point. Figure 4.3 shows all relevant relationships, classes and plugins related with this extension.

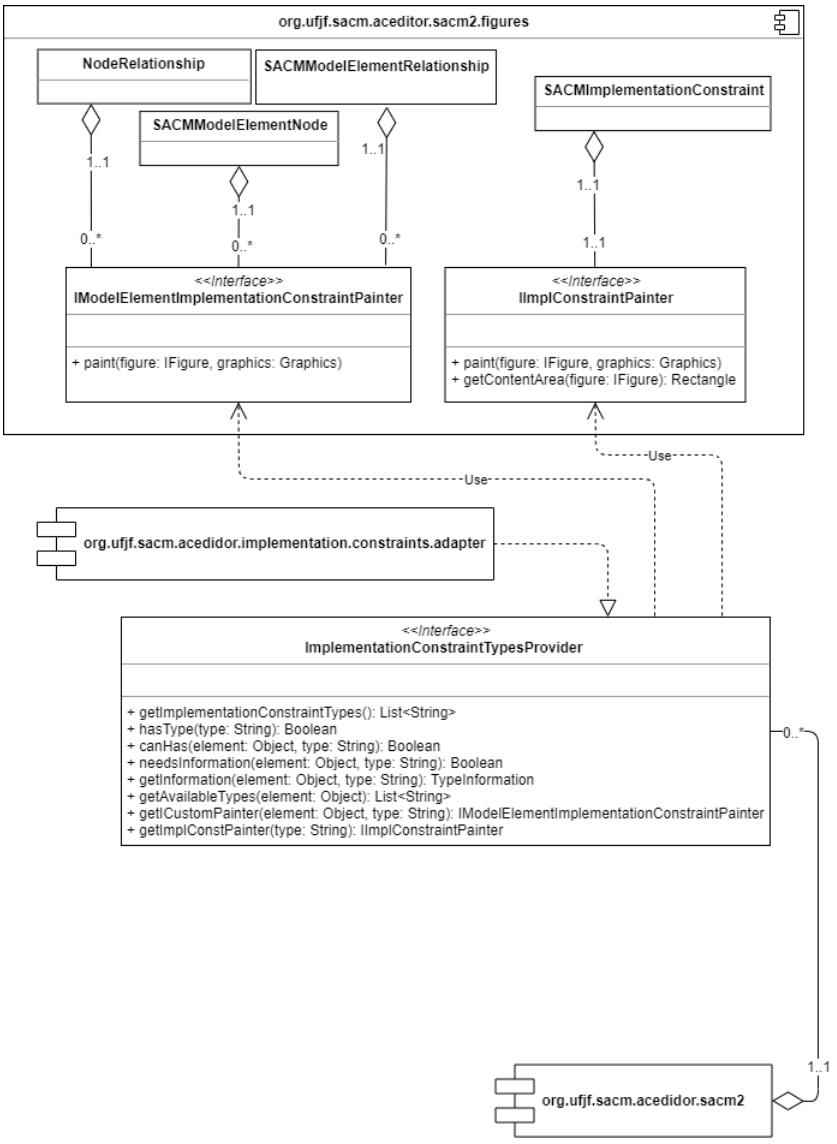

Figure 4.3: Extension for Provide Types of **ImplementationConstraint**

The problem how to represent **ImplementationConstraints** in a model element figure have been solved by some methods and the relationships with the following figure classes: the **NodeRelationship** is a figure class which represents all SACM relationships which ex-

tends from **ModelElement** and are nodos on diagram; the **SACMModelElementNode** is the figure super class which represents all the SACM elements which extend from **ModelElement** except for the relationships; and finally the **SACMModelElementRelationship** is the figure super class which represents the relationships between a **NodeRelationship** figure and a **SACMModelElementNode** figure. Thus with the method `getICustomPainter` of **ImplementationConstraintTypesProvider** gets the **IModelElementImplementationConstraintPainter** which is the custom drawn which will be drawn a on a **ModelElement** figure to represent that it has an **ImplementationConstraint** of a type. However this method dos not handle collisions, thus it is necessary know the drawn of all types **ImplementationConstraint** which can be added in that element, in order to implement a drawn which wont overlap the others. If the return of this method is `null` nothing will be drawn on the model element figure.

The problem how to represent **ImplementationConstraints** on the editor canvas, have been solved by the relationship between **SACMImplementationConstraint** figure class with the interface **IImplCostraitPainter**. The `getIConstPainter` method of **ImplementationConstraintTypesProvider** gets the **IImplCostraitPainter** to draw a custom figure for a **ImplementationConstraint** on the editor canvas. If the return of this method is `null` the **ImplementationConstraint** default figure will be drawn.

Finally, how to save the information required for a type of **ImplementationConstraint** in the model, e.g., a 'query' which finds in other model the value of an abstract **Term**, in order to allow the creation of a program able to get it. It have been solved by the methods of `needsInformation` and `getInformation` of **ImplementationConstraintTypesProvider**. the `needsInformation` method returns if in order to add a specific type of **ImplementationConstraint** in a element it is necessary more information, if it is necessary the `getInformation` is called. The `getInformation` return a **TypeInformation** which contains a complement and the information both strings. This information is saved in the **MultiLangString** content of **ImplementationConstraint** as an instance of **LangString**, the value of its *lang* property follows the template '$< Type >:< TypeComplement >$', e.g., 'type1:complement', and its *content* property contains the information required. Therefore, a standard way to embedd the pattern instantiation information into an **ImplementationConstraint** is suggested. This allows the instantiation program get such information, whatever this information and its language are.

The `hasType` method is used for selecting the **ImplementationConstraintType-**

sProvider implementation. Thus, the reason why two or more implementations of this extension point must not provide the same type(s) is that only one of them will be selected.

4.3.3 Validation Extension

The `org.ufjf.sacm.aceditor.sacm2.validation.ImplementationConstraintValidator` is the id of this extension point which an implantation of this extension point needs to use. From this extension must be create a new **Validator** whose the *Validator* field is the class which implements the **ImplementationConstraintValidator** interface.

The implementations of this extension point can provide validation rules for **ImplementationConstraint** with types. This validation works integrated with the validation plugin. Thus, when a **ImplementationConstraint** has a type the validation plugin search for its rules/restrictions among all existing implementations of this extension. The problem of establishing a way to manage potential validation rules over the types of constraints and their possible instantiation information has been solved by this extension point, this problem is presented in Section 2.7.2 which discuss an instantiation program for the SACM. Figure 4.4 shows all relevant relationships, classes and plugins related with this extension.

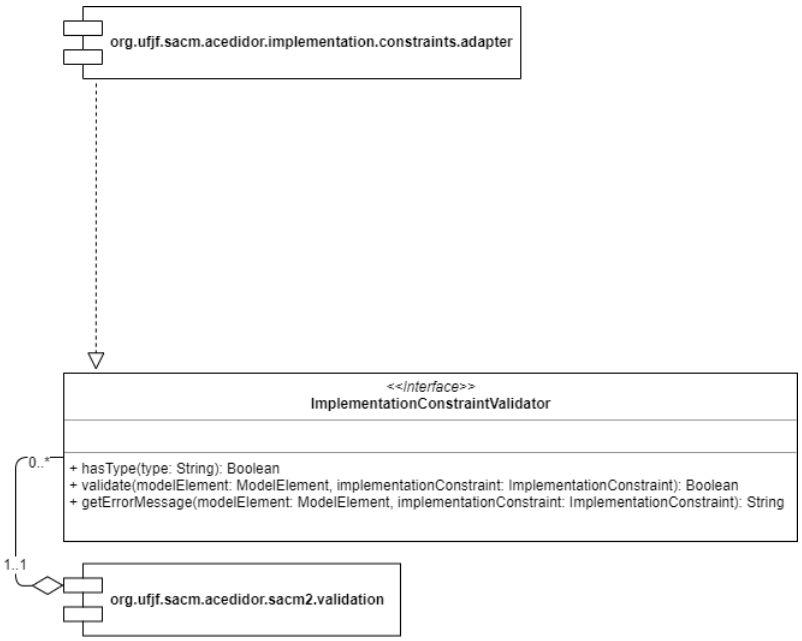

Figure 4.4: Extension for Validate **ImplementationConstraint** With a Type

The method `validate` of **ImplementationConstraintValidator** validate a **ModelElement** with a **ImplementationConstraint** with a type. If the return of `validate` method is `false` for a **ModelElement**, then the method `getErrorMessage` is called in order to get a

message to show the users why the validation failed.

The validation plugin extension point may have various implementations. However, as 'Provider Extension' the `hasType` method is used for selecting the **ImplementationConstraintValidator** implementation. Thus, the reason why two or more implementations of this extension point must not validate the same type(s) is that only one of them will be selected to validate the types(s).

5 Assurance Case Patterns

This chapter presents the patterns that have been used to specify the case studies. Section 5.1 explores the Hazard Avoidance pattern, Section 5.2 the Risk Argument pattern, Section 5.3 the Absence of Hazardous Software Failure Mode pattern and Section 5.4 the Artifact Argument pattern.

5.1 Hazard Avoidance Pattern

Figure 5.1 shows the GSN representation of this pattern and Figure 5.2 shows the equivalent representation of it in the SACM ACEditor. The hazard avoidance pattern (KELLY; MCDERMID, 1997) decomposes the augment that the system is acceptable safe (SysSafe) into sub-claims arguing over the risk posed by each system hazards (RiskHzdX) is acceptable. This argumentation strategy (ArgOverRiskHzds) is in the context of the identified system hazards(IdentHzds). This whole argumentation is in the context the system properties as definition (SystemDefn) and environment (Environment). This argumentation has also a target safety standard, which it depends on to be acceptable or not acceptable as safe.

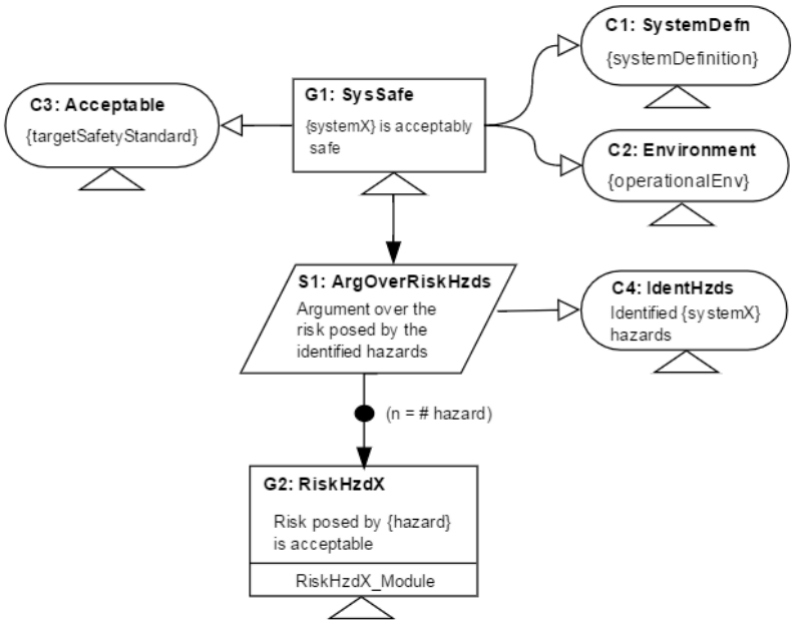

Figure 5.1: Hazard Avoidance Pattern in GSN

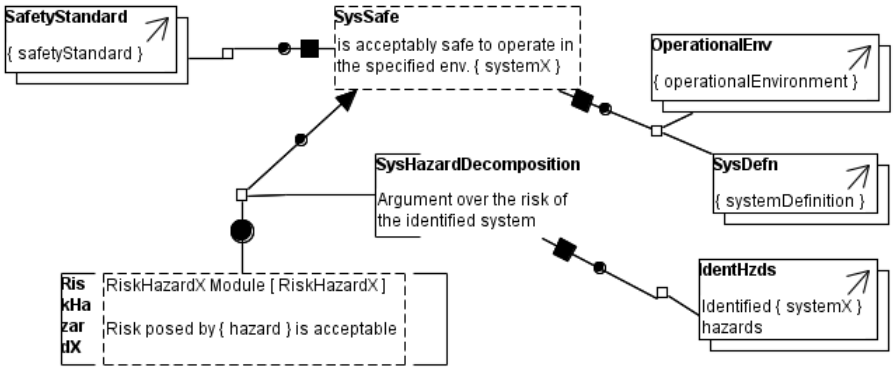

Figure 5.2: Hazard Avoidance Pattern in SACM

5.2 Risk Argument Pattern

This pattern argues 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 5.3 shows the GSN representation of this pattern and Figure 5.4 shows the equivalent representation of it in the SACM ACEditor. 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 TLFailureCondition. 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.

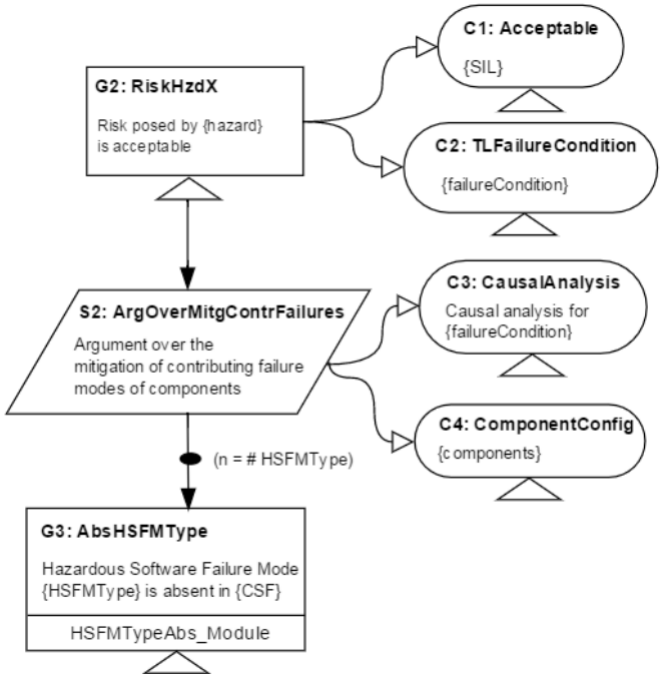

Figure 5.3: Risk Argument Pattern in GSN

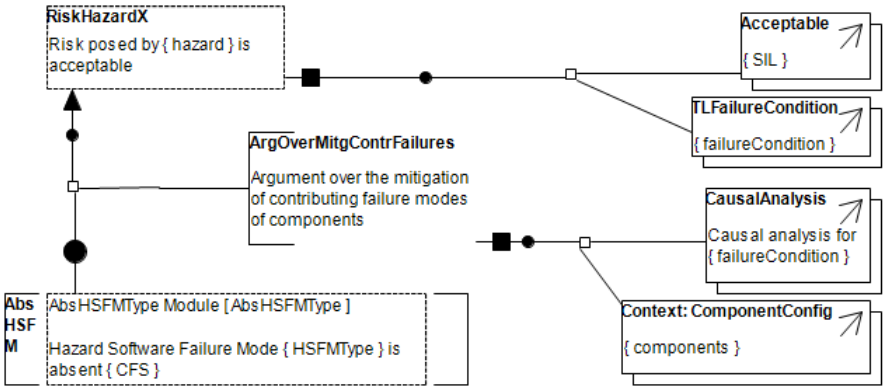

Figure 5.4: Risk Argument pattern in SACM

5.3 HSFM Pattern

An Absence Hazardous Software Failure Mode (HSFM) fault mitigation pattern argues 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 5.5 shows the GSN representation of this pattern and Figure 5.6 shows the equivalent representation of it in the SACM ACEditor. This pattern decomposes the claim ABSHSFMType into tree sub-claims: *i)* AbValPrimary arguing that the current failure mode is acceptable; *ii)* AbValSecondary arguing that the failure modes of other components that

contribute to the current failure mode are acceptable; *iii)* AbTypeControl arguing that the contributory software functionality component is scheduled and allowed to run once. The AbValSecondary 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 other "Absence Hazardous Software Failure Mode" (HSFM) fault mitigation argument.

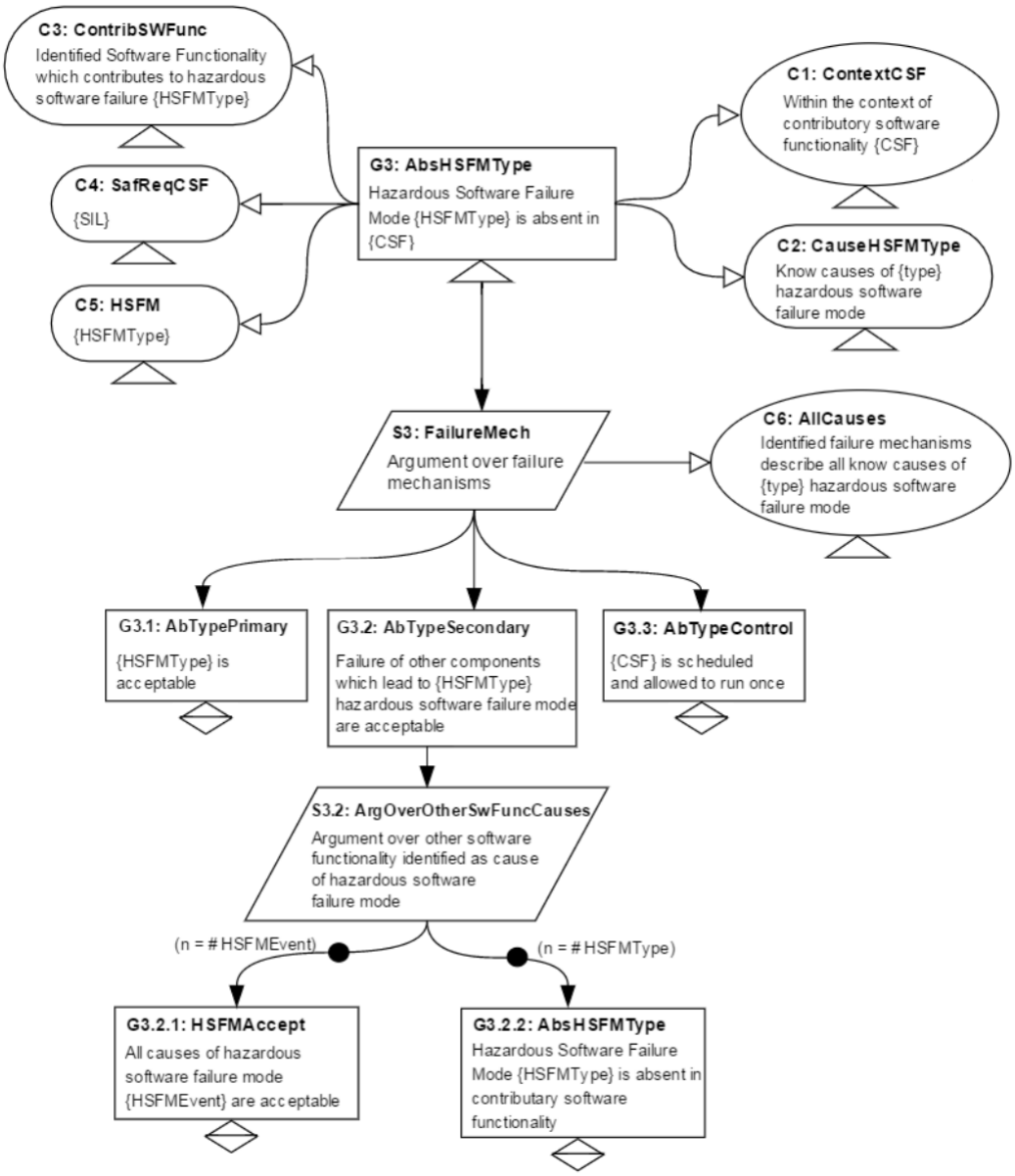

Figure 5.5: HSFM pattern in GSN

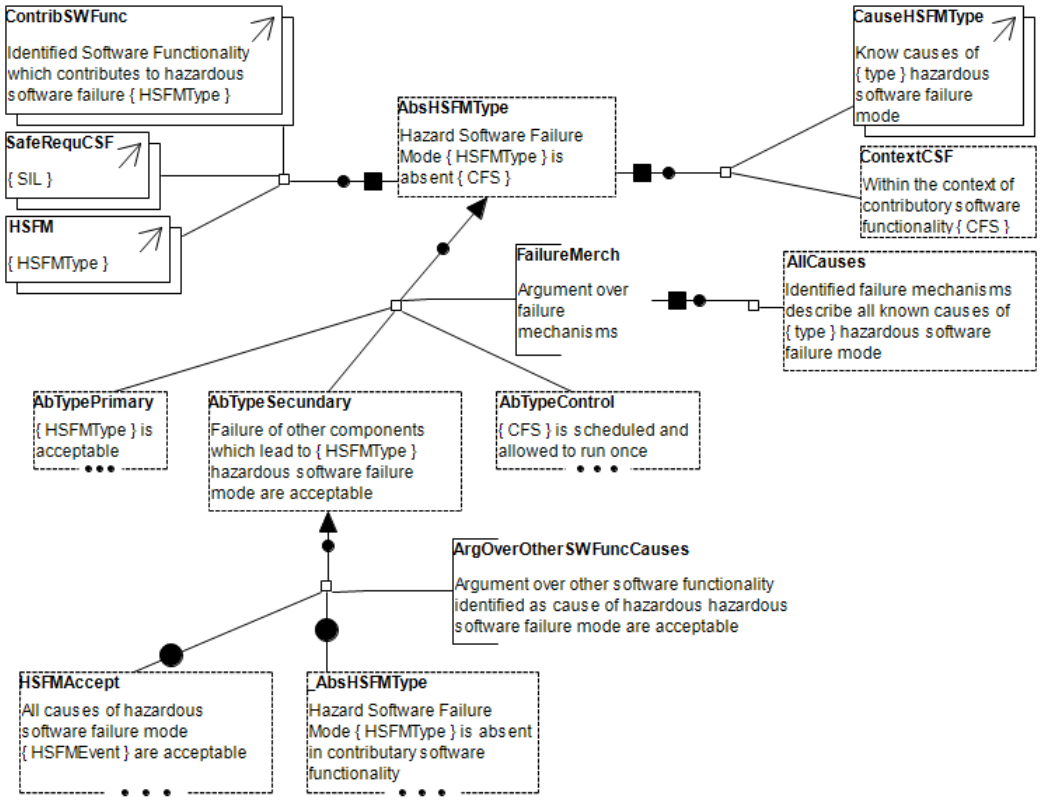

Figure 5.6: HSFM Pattern in SACM

5.4 Functional Hazard Assessment Pattern

The OMG-SACM provides support for the specification of artifact provenience, e.g., where an artifact came from and who is involved with an artifact, which GSN does not provide. Thus, artifact patterns related to product requirements, architecture, functional hazard assessment, fault trees, and FMEA results, can be created with the support of the SACM 2.1 model editor.

Figure 5.7 shows the Functional Hazard Assessment artifact pattern specification in the SACM ACEditor. This pattern defines that a Hazard Analysis artifact is resultant from performing the Hazard Identification activity with the support of a given Hazard Analysis Technique. A Hazard Analysis artifact can be owned by one or more participants involved in the execution of Hazard Analysis activities. Finally, a Hazard Analysis artifact is associated with a Hazard Analysis File resource generated at the end of the hazard analysis process, (OLIVEIRA, 2016).

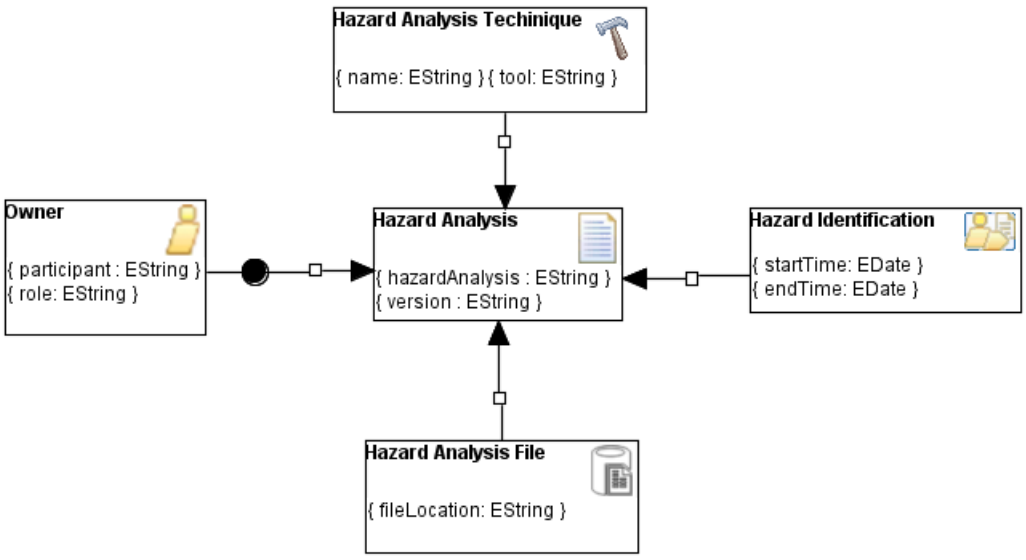

Figure 5.7: Functional Hazard Assessment Pattern in SACM

It is necessary to be able to cite artifacts that provide supporting evidence, context, or additional description within an argument structure. Thus, **ArtifactReference** element of SACM allows there to be an objectified citation of this information within the structured argument, thereby allowing the relationship between this artifact and the argument to also be explicitly declared, (OMG, 2019).

6 Case Study of Hybrid Breaking System

This chapter contains the case study of the Hybrid Breaking System (HBS). Section 6.1 contains a brief architecture of HBS explanation. The assurance cases patterns instantiated for this case study are described in Section 6.2

6.1 Architecture of HBS

The objective of this system is the integration in electrical vehicles, particularly for propulsion architectures that comprise one electrical motor per wheel, (FREITAS; CASTRO; ARAUJO, 2011). This system is called hybrid because the braking is achieved by the combined action of electrical In-Wheel Motors (IWMs), and frictional Electromechanical Brakes (EMBs). The IWMs work as generators and transform the vehicle kinetic energy into electrical energy which charges the Powertrain Battery, thus it increases the vehicle's range. Omission or incorrect braking torque failures in the wheels while braking may lead to catastrophic consequences for the driver, thus the HBS configurations should not cause them.

Figure 6.1 shows the general architecture of HBS. MechElecPedal is the hardware that processes the actions from the mechanical pedal and captures the driver presses. The CommunicationBus components represent a duplex bus communication software that sends braking torque to the wheel-braking units. AUXBattery is hardware that provides power to the brake units while braking. PWTBattery is also hardware that stores the electrical energy produced by the in-wheel motors (IWMs).

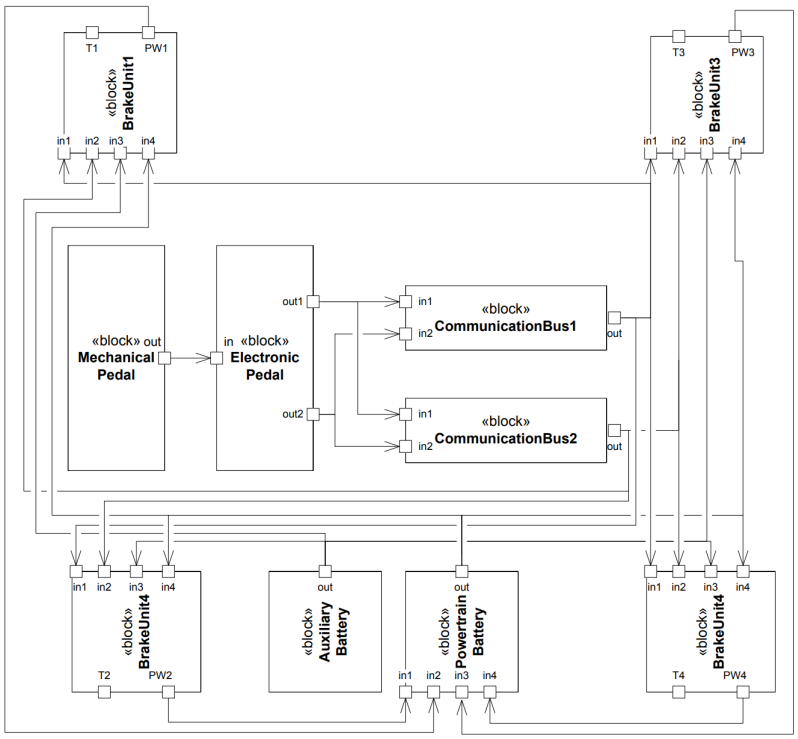

Figure 6.1: Architecture of HBS

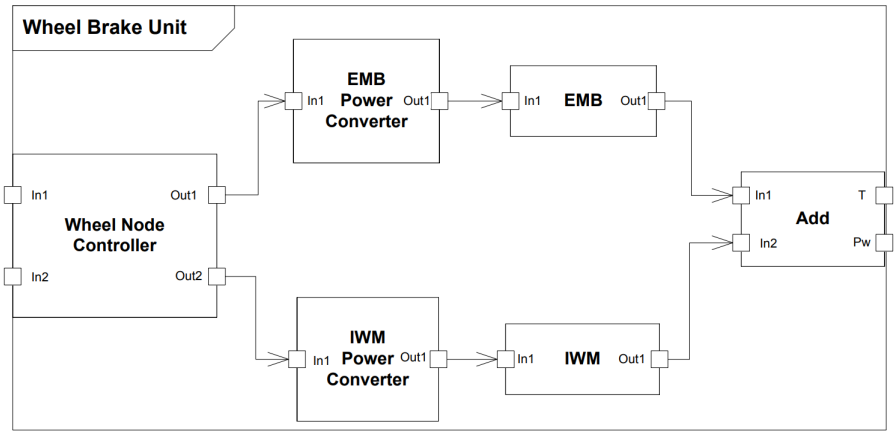

Figure 6.2: Architecture of Wheel Break Unit of HBS

Figure 6.2 shows the internal components of a Wheel Break Unit. Each wheel brake unit contains a Wheel Node Controller (WNC) which calculates the amount of braking torque to be produced by each wheel braking actuator, and it sends commands to Electromechanical Braking (EMB) and In-Wheel Motors (IWM) power converters, they are responsible to control the EMB and the IWM braking actuators. While braking, the electric power flows from the AUXBattery to EMB via EMB Power Converter, and IWM acts as a power generator providing energy for the Powertrain Battery via IWM Power Converter.

6.2 SACM Assurance Cases

In this section, the assurance case patterns which have been presented in Chapter 5 are instantiated for the HBS system. Although these instances do not cover all the HBS, the application of all patters are presented.

6.2.1 Overview

Figure 6.3 shows an overview of the instantiated assurance case patterns. This overview describes how these patterns are related. It contains only the parts of the assurance case which will be explored separately in the next sections.

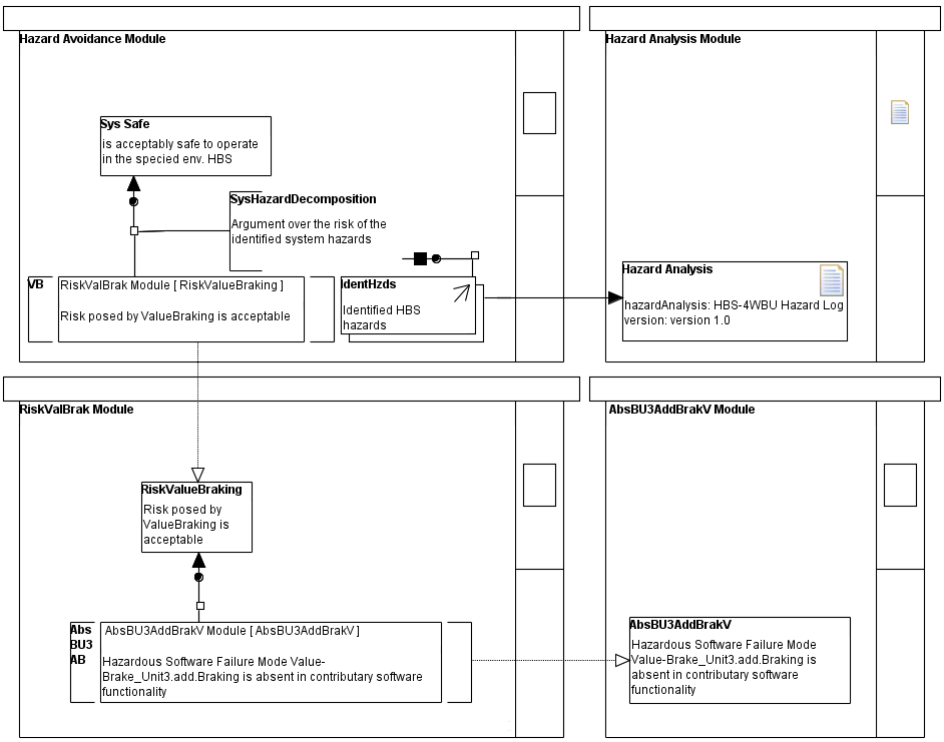

Figure 6.3: HBS Assurance Case Overview

The VB cites the RiskValueBraking and AbsBU3AB cites the AbsBU3AddBrakV. The IdentHzds has the Hazard Analysis as one of its referenced artifact elements.

6.2.2 Hazard Avoidance

The Hazard Avoidance Module decomposes the claim arguing that the HBS system is acceptably safe to operate in the four-wheel brake units environment, into two sub-claims VB and NB4W arguing that the risk posed by each hazard is acceptable. Each of these sub-claims cites claims

which are encapsulated in a separated risk argument module. Figure 6.4 shows the internal vision of Hazard Avoidance Module.

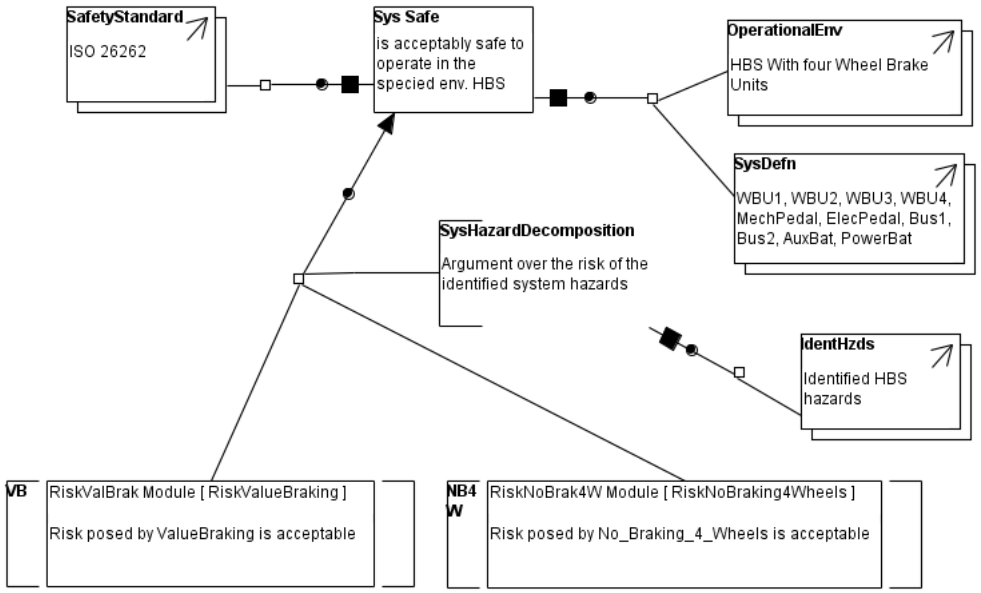

Figure 6.4: HBS Hazard Avoidance

6.2.3 Risk Argument

The RiskValueBraking top-level claim is stated in the context of ASIL "D" allocated to "Value Braking" system hazard in Acceptable, and the top-level failure condition leading to this hazard in claim TLFailureCondition. The top-level claim is decomposed into sub-claims arguing the mitigation of component failures that directly contribute to the occurrence of "Value Braking" hazard (ArgOverMitgContrFailures), in this case, incorrect values in "Brake_Unit1.Add" (AbsBU1AB) and "Brake_Unit3.Add" (AbsBU3AB) component outputs, which can cause an incorrect braking torque while braking. Such decomposition strategy is defined in the context of the causal chain defined in the "Value Braking" fault tree. AbsBU1AB and AbsBU3AB cites claims in AbsBU1AddBrakV Module and AbsBU3AddBrakV Module respectively, which are supported by sub-claims arguing the absence of each contributing hazardous software failure mode. Figure 6.5 shows the internal vision of RiskValBrak Module.

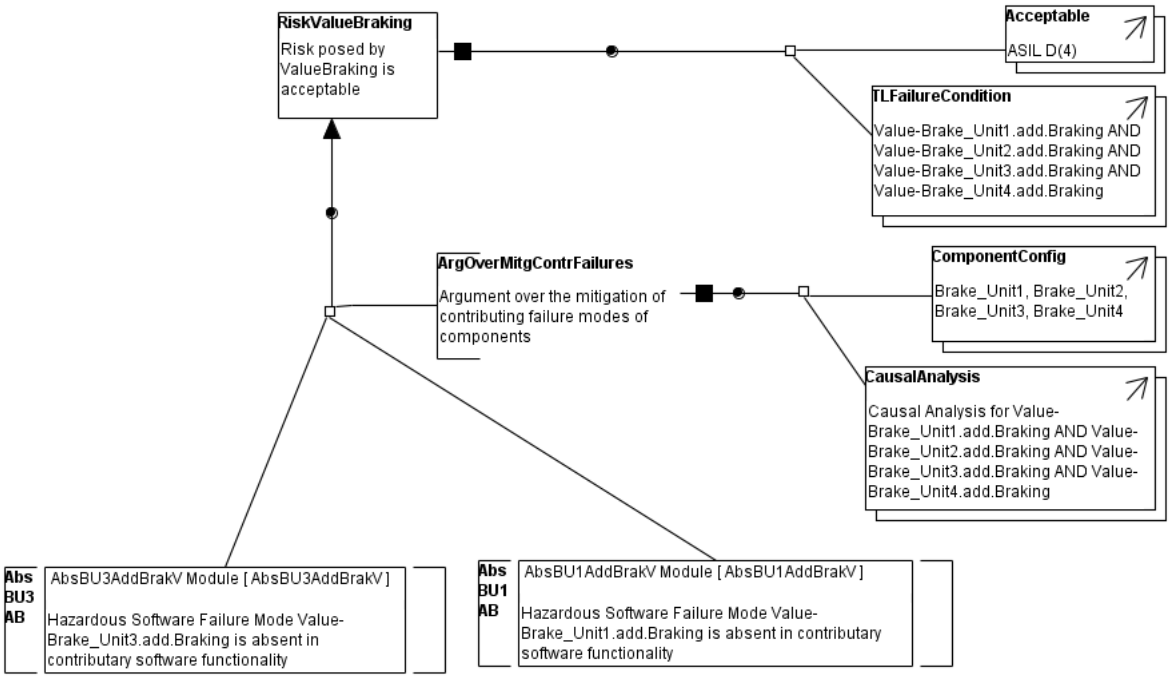

Figure 6.5: HBS Risk Argument

6.2.4 HSFM

The AbsBU3AddBrakV Module decomposes the claim AbsBU3AddBrakV into sub-claims: The AbValPrimary argues that an internal failure in "Brake_Unit3.Add" component is acceptable; AbValSecondary argues that the failure modes of other components that contribute to incorrect value of "Brake_Unit3.Add.Braking" output port are acceptable ; and AbTypeControl arguing that the "Brake_Unit3.Add" component is scheduled and allowed to run once". The AbValSecondary is further decomposed into fault mitigation sub-claims arguing that incorrect values in "Brake_Unit3.EMB_Power_Converter" and "Brake_Unit3EMB" components are acceptable.

BU3EMBPowerConvVFailure1Accept and BU3EMBVFailure1Accept claims argue that all causes of each failure event specified in fault tree leaf nodes do not lead the system to an unsafe state. The claim AbValSecondary is also decomposed into other "Absence Hazardous Software Failure Mode" fault mitigation argument modules, e.g., the AbsBU3NCO1 argument module. Figure 6.6 shows the internal vision of AbsBU3AddBrakV Module.

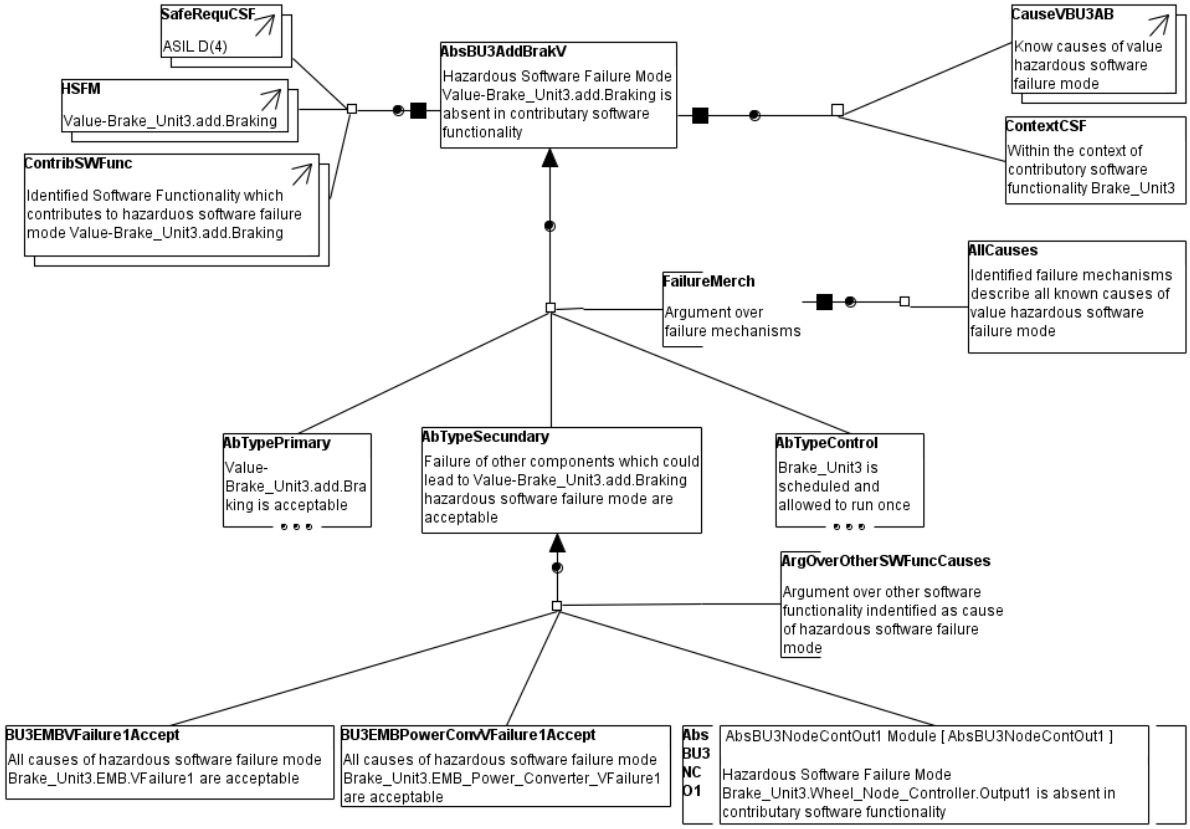

Figure 6.6: HBS HSFM Assurance Case

6.2.5 Functional Hazard Assessment

The Hazard Analysis Module artifact package, illustrated in Figure 6.7, provides the provenance for the hazard log artifact referenced by IdentHzd within Hazard Avoidance Module. This artifact is owned by two safety analysts who developed it with the support of "HaZard and OPerability Study analysis" technique and "HiP-HOPs" compositional safety analysis tool. The hazard log artifact was created during the Hazard Identification activity initiated by the owners in "10 Set of 2015" and finished in "14 Set 2015". The hazard log artifact is available in the form of hyperlinked web pages. Figure 6.7 shows the internal vision of Hazard Analysis Module.

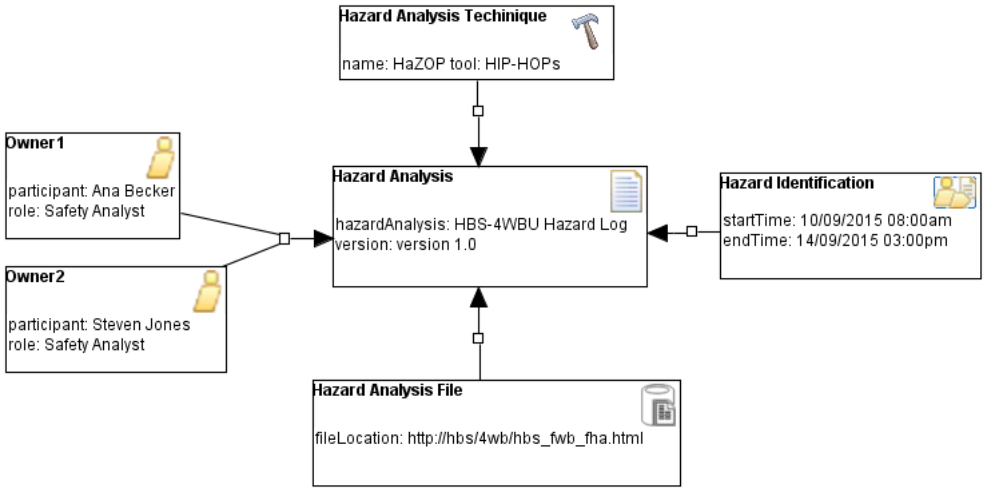

Figure 6.7: HBS Functional Hazard Assessment

7 Conclusion

This chapter contains the contributions of this work. It also discusses some research directions.

7.1 Contributions

The most important contribution of this work is the development of an assurance case editor in compliance with the OMG SACM standard, the SACM ACEditor is available at GitHub[1]. The graphical representation of some elements has been developed in SACM ACEditor, because SACM 2.1 meta-model does not provide any graphical representation for the elements beyond those within argumentation metamodel. Thus, this graphical representation can contribute to the development of the SACM graphical notation.

The type of property added to **ImplementationConstraint** elements provide a way to mark the code/query that should be used for automatic instantiation into their multi-language *content*, which is another contribution of this work. GSN pattern extensions have their semantics, syntax, and representations for constraints. With the 2 extension points of the developed editor, some of the GSN constraints have been incorporated into the SACM ACEditor. The tool supports the visualisation of **multiplicity**, **optional** and **choices** as GSN notation. Therefore, the representation of GSN constraints in SACM ACEditor is also a contribution of this work.

The other contribution of this work is the development of a model transformation plugin to support GSN to SACM model transformations. The transformation reduces the cost of creating SACM models from GSN models. Although it only does the transformation of models developed in a specific GSN editor, the transformation from other GSN models developed in other editors can be done indirectly. The description of the process followed by the author for creating the EMF-based editors is also a contribution of this work. Tool developers can follow this process to build modeling tools for their domain-specific languages.

Some of the contributions can contribute to the development of an automatic instantiation program. However, the SACM Aceditor does not provide support for the automatic assurance case pattern instantiation. Another limitation is the lack of experimental studies to validate the developed editor in an industrial context.

[1] https://github.com/LuisFelipeAN/SACM-ACEditor

7.2 Research Directions

The Creation of a generic instantiation program for the SACM is one of the research directions. This program can get the instantiation query/information of **ImplementationConstrains** and creating the model from an assurance case pattern. However, this depends on **Implementation-Constrains** and where they must be to make possible the automatic instantiation. Therefore, the definition of the types of **ImplementationConstrains** that could lead this instantiation is the other research direction.

Bibliography

ACWG. *Goal Structuring Notation Community Standard (Version 2)*. [S.l.]: SCSC, 2018. Available at: <https://scsc.uk/scsc-141B>. Access on: May 15^{th}, 2019.

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

BRANCO, K. R. et al. Tiriba-a new approach of uav based on model driven development and multiprocessors. In: IEEE. *2011 IEEE International Conference on Robotics and Automation*. [S.l.], 2011. p. 1–4.

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

ECLIPSE. *Epsilon*. 2018. Available at: <https://www.eclipse.org/epsilon/>. Access on: May 22^{th}, 2018.

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

EPSILON. *Customizing a GMF editor generated by EuGENia*. Available at: <https://www.eclipse.org/epsilon/doc/articles/eugenia-polishing/>. Access on: Jun 7^{th}, 2019.

EPSILON. *Example: Use Epsilon in standalone Java applications*. Available at: <https://www.eclipse.org/epsilon/examples/index.php>. Access on: Jun 11^{th}, 2019.

EPSILON. *Live validation and quick-fixes in GMF-based editors with EVL*. Available at: <https://www.eclipse.org/epsilon/doc/articles/evl-gmf-integration/>. Access on: Jun 11^{th}, 2019.

FREITAS, D.; CASTRO, R. de; ARAUJO, R. E. Hybrid abs with electric motor and friction brakes. 2011.

GSN. *Goal Structuring Notation*. 2018. Available at: <http://www.goalstructuringnotation.info/>. Access on: November 14^{th}, 2018.

HAILPERN, B.; TARR, P. Model-driven developmen: The good, the bad, and the ugly. *IBM systems journal*, v. 45, n. 3, p. 541–561, 2006.

HAWKINS, R. et al. Weaving an assurance case from design: a model-based approach. In: IEEE. *High Assurance Systems Engineering (HASE), 2015 IEEE 16th International Symposium on*. [S.l.], 2015. p. 110–117.

JOHNSON, L. A. et al. Do-178b, software considerations in airborne systems and equipment certification. *Crosstalk, October*, v. 199, 1998.

KANCHANA, S.; FANEY, J. Study of safety management by using gis in coimbatore. *International Journal of Scientific & Technology Research*, v. 4, n. 8, p. 367–370, 2015.

KELLY, T.; WEAVER, R. The goal structuring notation–a safety argument notation. In: CITESEER. *Proceedings of the dependable systems and networks 2004 workshop on assurance cases*. [S.l.], 2004. p. 6.

KELLY, T. P.; MCDERMID, J. A. Safety case construction and reuse using patterns. In: *Safe Comp 97*. [S.l.]: Springer, 1997. p. 55–69.

KOLOVOS, D. et al. *The Epsilon Book*. 2013. Available at: <https://www.eclipse.org/epsilon/doc/book/>. Access on: May 20^{th}, 2018.

KOLOVOS, D. S. et al. Bridging the epsilon wizard language and the eclipse graphical modeling framework. In: *Modeling Symposium, Eclipse Summit Europe, Ludwigsburg, Germany*. [S.l.: s.n.], 2007.

KOLOVOS, D. S. et al. Taming emf and gmf using model transformation. In: SPRINGER. *International Conference on Model Driven Engineering Languages and Systems*. [S.l.], 2010. p. 211–225.

LEVESON, N. White paper on approaches to safety engineering. *Disponible en ligne sur le site de l'auteur (sunnyday. mit. edu/caib/concepts. pdf)*, 2003.

MELLOR, S. J.; CLARK, T.; FUTAGAMI, T. Model-driven developmen: guest editors' introduction. *IEE software*, v. 20, n. 5, p. 14–18, 2003.

OLIVEIRA, A. L. d. *A model-based approach to support the systematic reuse and generation of safety artefacts in safety-critical software product line engineering*. Thesis (Phd) — Universidade de São Paulo, 2016.

OLIVEIRA, A. L. D. et al. Variability management in safety-critical systems design and dependability analysis. *Journal of Software: Evolution and Process*, Wiley Online Library, v. 31, n. 8, p. e2202, 2019.

OMG, O. M. G. *Structured Assurance Case Metamodel (SACM)*. 2019. Available at: <https://www.omg.org/spec/SACM/2.1/Beta1/PDF>. Access on: May 20^{th}, 2019.

SAE. *Guidelines for Development of Civil Aircraft and Systems ARP4754A*. 2010. Available at: <https://www.sae.org/standards/content/arp4754a/>. Access on: May 22^{th}, 2018.

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

SOMMERVILLE, I. *Engenharia de Software*. 6. ed. [S.l.]: Peterson Education, 2003.

SOMMERVILLE, I. *Engenharia de Software*. 9. ed. [S.l.]: Peterson Education, 2011.

STEINBERG, D. et al. *EMF: eclipse modeling framework*. 2. ed. [S.l.]: Peterson Education, 2008.

UP-TIME, I. P. Safety life-cycle. Citeseer, 2007.

WEI, R. et al. Model based system assurance using the structured assurance case metamodel. *Journal of Systems and Software*, Elsevier, 2019.

WRIGHT, J. *GMF Diagram Partitioning*. 2005. Available at: <https://jevon.org/wiki/GMF_Diagram_Partitioning>. Access on: Jun 11^{th}, 2019.

A Case Study of Tiriba Flight Control

This chapter contains the case study of Tiriba. Section A.1 contains a brief architecture of Titriba explanation. The assurance cases patterns instantiated for this case study are described in Section A.2.

A.1 Architecture of TFC

The Tiriba is a small autonomous electrical airplane used into pre-defined missions and applications, e.g, agricultural and environmental monitoring, (BRANCO et al., 2011). However, this case study focus on part of this system, the Tiriba Flight Control (TFC) which was presented by (OLIVEIRA, 2016) and (OLIVEIRA et al., 2019). The TFC consists of a control subsystem that aims to start the flight mode, process, and setup of flight commands, keep flight conditions and execute commands sent by the navigation subsystem, (OLIVEIRA et al., 2019). It has four types of pilots autonomous, auto, assisted and manual, the components of these types are highlighted in Figure A.1.

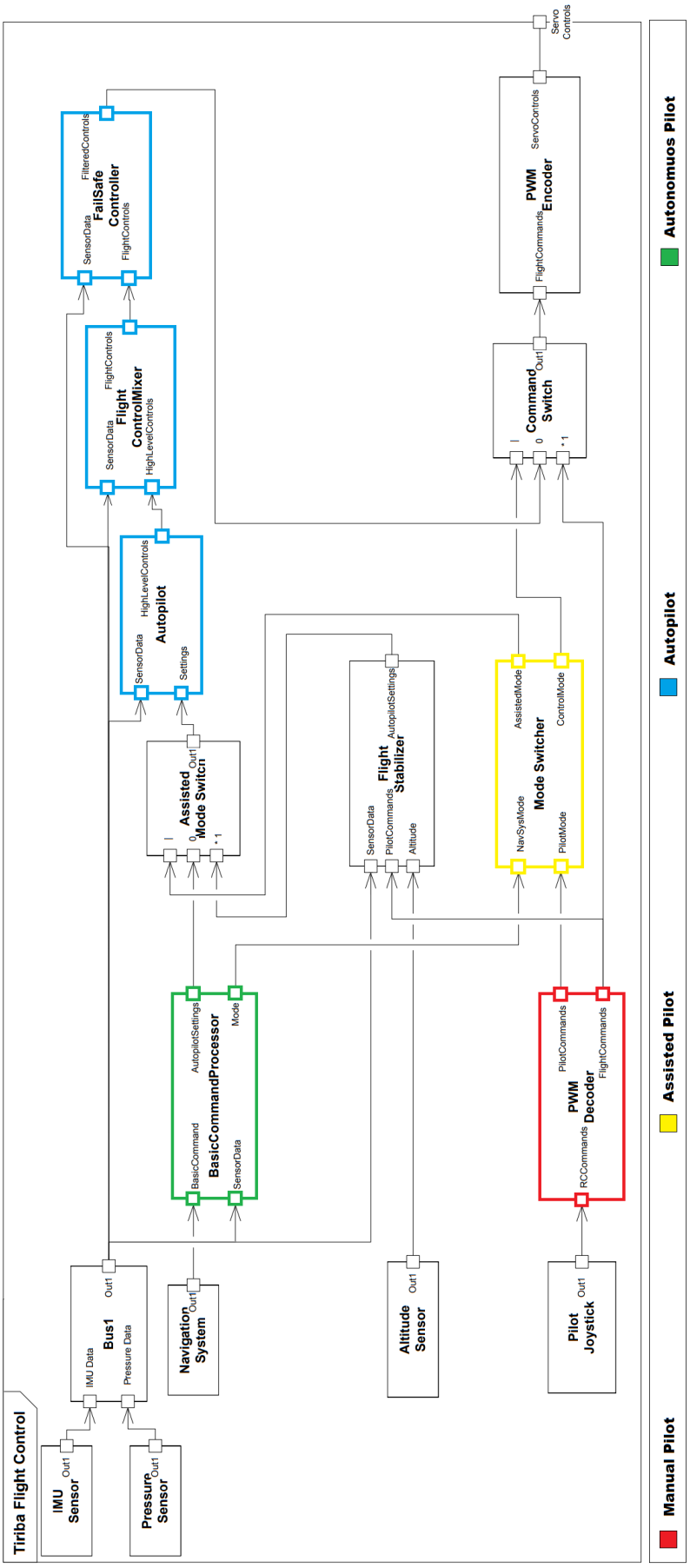

Figure A.1: Architecture of TFC

A.2 Tiriba Assurance Cases

A.2.1 Overview

Figure A.2 shows an overview of the instantiated assurance case patterns. This overview describes how these patterns are related. It contains only the parts of the assurance case which will be explored separately in the next sections.

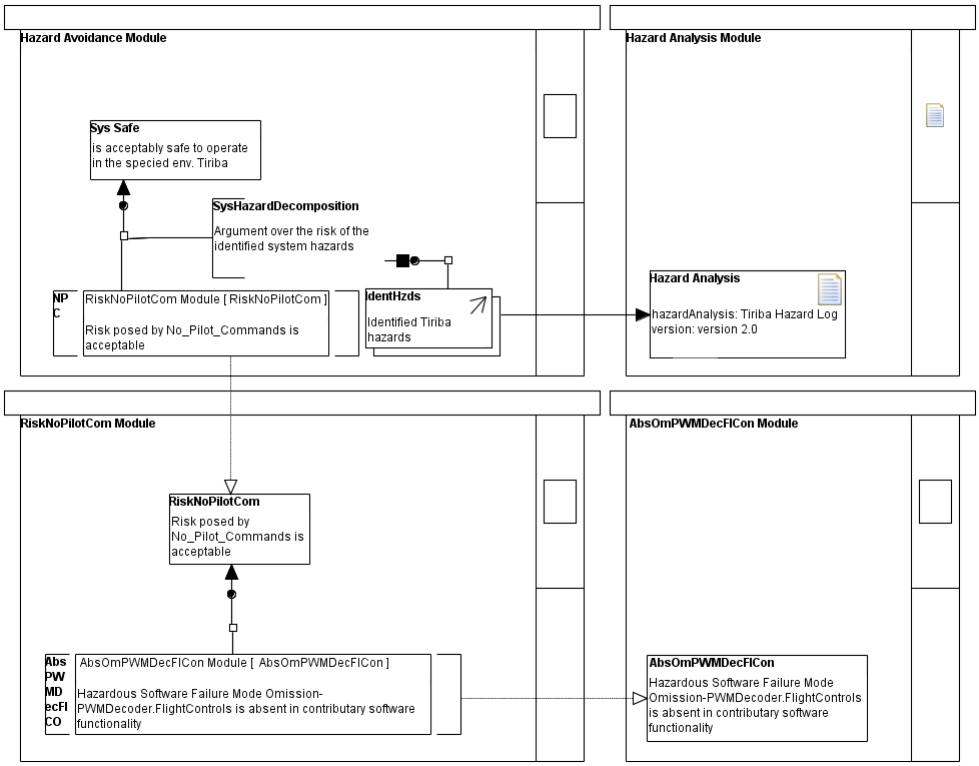

Figure A.2: Tiriba Assurance Case Overview

The NPC claim cites the RiskNoPilotCom and AbsPWMDecFlCO claim cites the AbsOmPWMDecFlcon. The IdentHzds has the Hazard Analysis as one of its referenced artifact elements.

A.2.2 Hazard Avoidance

The Hazard Avoidance Module decomposes the claim arguing that Tiriba system is acceptably safe to operate in the all pilots' environment, into two sub-claims NPC and VPC arguing that the risk posed by each hazard is acceptable. Each of these sub-claims cites claims which are encapsulated in a separated risk argument module. Figure A.3 shows the internal vision of Hazard Avoidance Module.

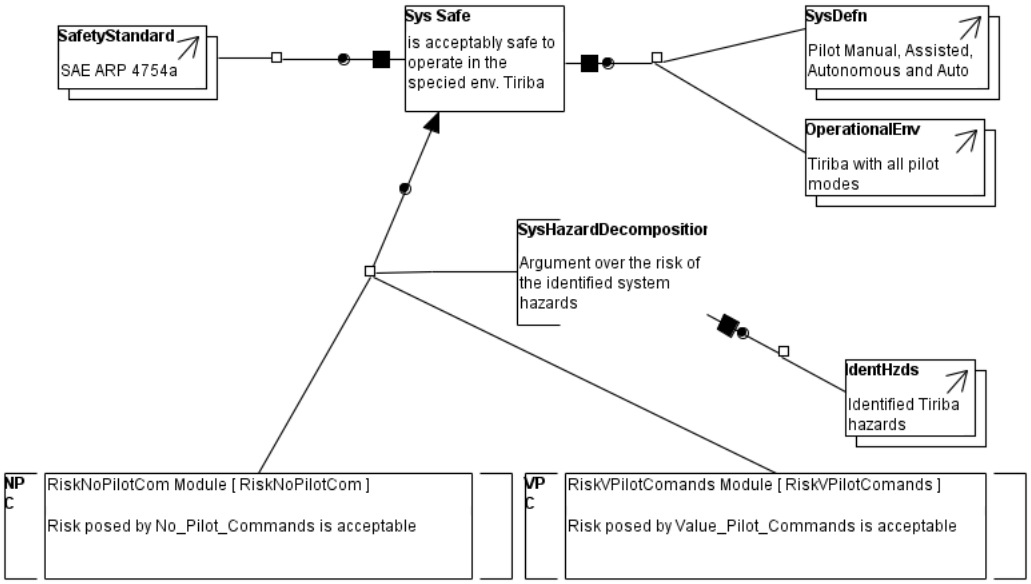

Figure A.3: Tiriba Hazard Avoidance

A.2.3 Risk Argument

The RiskNoPilotCom top-level claim is stated in the context of DAL "A(4)" allocated to "No_Pilot_Commands" system hazard in Acceptable, and the top-level failure condition leading to this hazard in claim TLFailureCondition. The top-level claim is decomposed into sub-claims arguing the mitigation of component failures that directly contribute to the occurrence of "No_Pilot_Commands" hazard (ArgOverMitgContrFailures), in this case, omission failures in "PWMDecoder" (AbsPWMDecFLCO) and "FailSafeController" (AbsFCO) component outputs, which contribute to the occurrence of the omission of UAV flight control commands. Such decomposition strategy is defined in the context of the hazard causal chain defined in the "No_Pilot_Commands" fault tree. AbsFCO and AbsPWMDecFLCO cites claims in other modules, AbsFSCFilComOmission Module and AbsOmPWMDecFlCom Module respectively, which are supported by sub-claims arguing the absence of each contributing hazardous software failure mode. Figure A.4 shows the internal vision of RiskNoPilotCom Module.

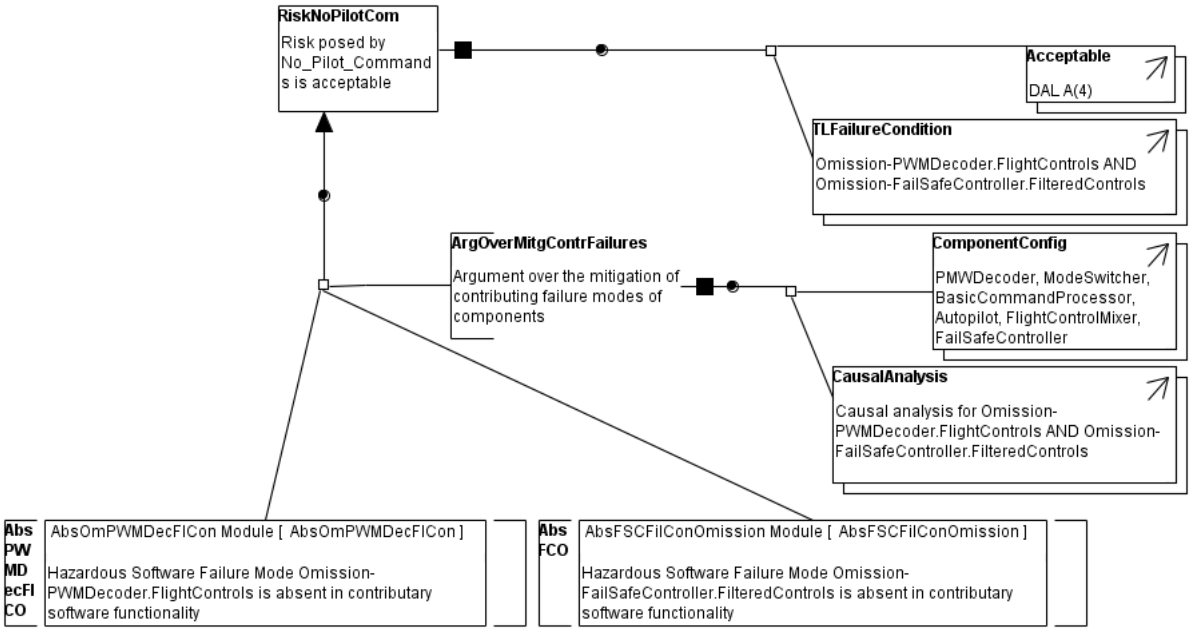

Figure A.4: Tiriba Risk Argument

A.2.4 HSFM

The AbsOmPWMDecFlCom Module decomposes the claim AbsOmPWMDecFlCom into sub-claims: AbValPrimary arguing that an internal failure in "PWMDecoder component is acceptable"; AbValSecondary arguing that the failure modes of other components that contribute to omission of "PWMDecoder.FlightControls" output port are acceptable ; and AbTypeControl arguing that the "PWMDecoder" component is scheduled and allowed to run once. The AbValSecondary is further decomposed into fault mitigation sub-claims arguing that omission failures in "BusCreator1" and "PilotJoystick" components are acceptable. BusCreator1Accept and PilotJoystickAccept claims argue that all causes of each failure event specified in fault tree leaf nodes do not lead the system to an unsafe state. Figure A.5 shows the internal vision of AbsPWMDecFlCon Module.

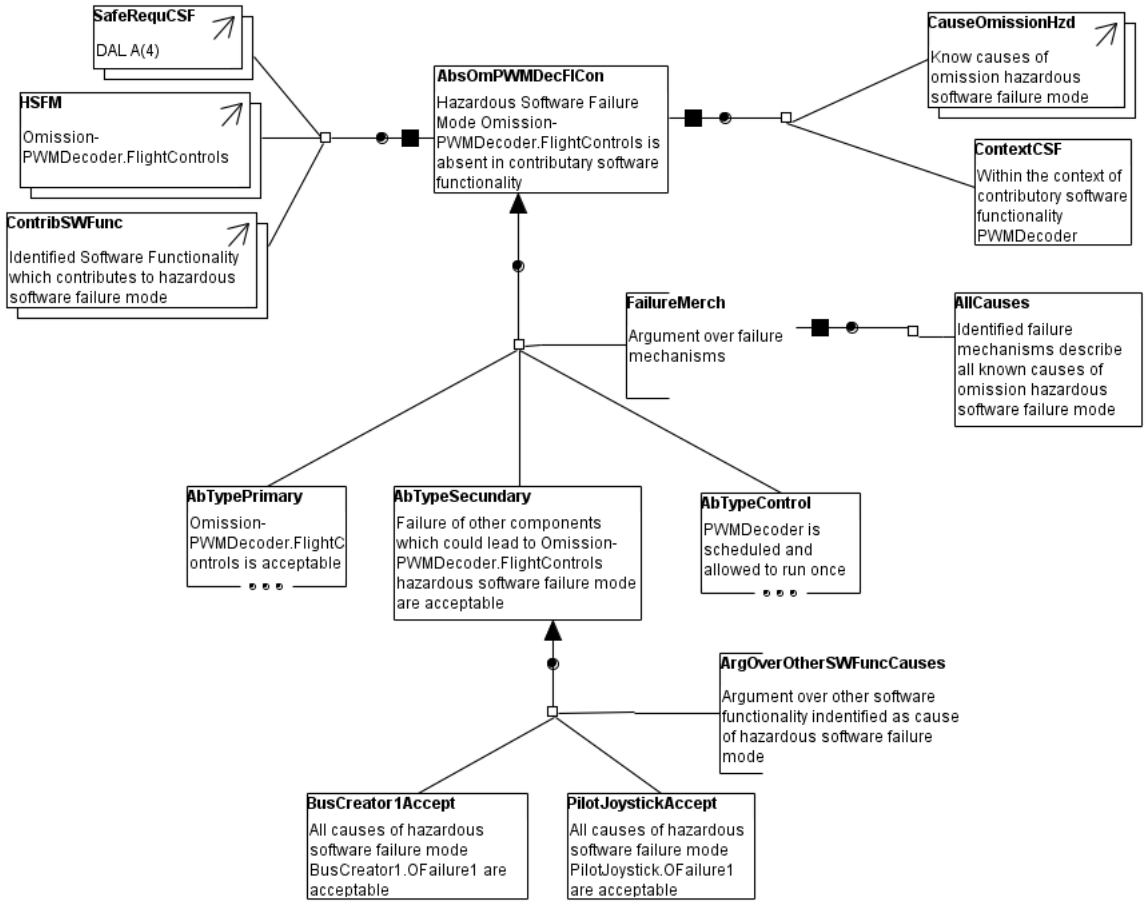

Figure A.5: Tiriba HSFM Assurance Case

A.2.5 Functional Hazard Assessment

The Hazard Analysis Module artifact package, illustrated in Figure A.6, provides the provenance for the hazard log artifact referenced by IdentHzd within Hazard Avoidance Module. This artifact is owned by two safety analysts who developed it with the support of "HaZard and OPerability Study analysis" technique and "Osate-AADL" tool. The hazard log artifact was created during the Hazard Identification activity initiated by the owners in "15 Set of 2015" and finished in "26 Set 2015". The hazard log artifact is available in the form of hyperlinked web pages. Figure A.6 shows the internal vision of Hazard Analysis Module.

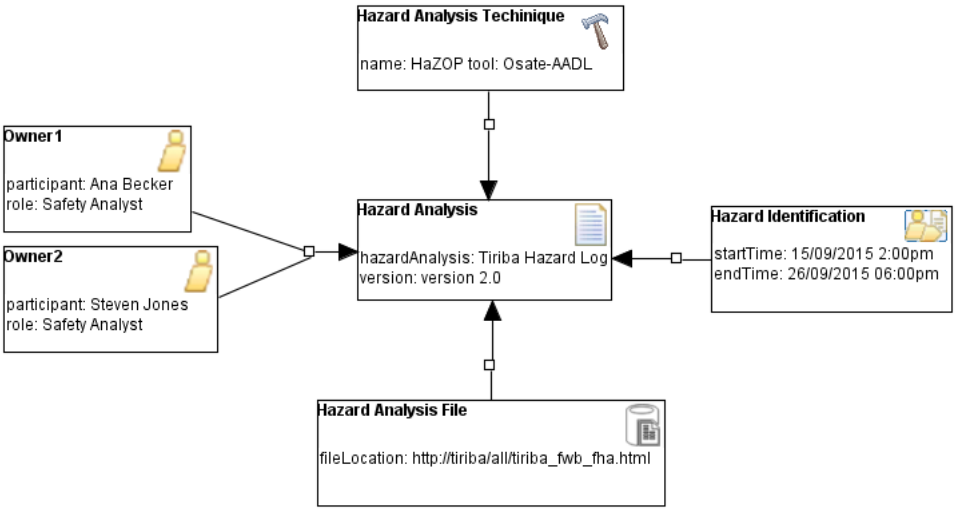

Figure A.6: Triba Functional Hazard Assessment