A TOOL-SUPPORTED METHOD FOR FALLACIES DETECTION IN PROCESS-BASED ARGUMENTATION

Laura Gómez Rodríguez
lgz17001@student.mdh.se

Examiner: Barbara Gallina
Mälardalen University, Västerås, Sweden

Examiner: Radu Dobrin
Mälardalen University, Västerås, Sweden

Supervisor: Faiz UL Muram
Mälardalen University, Västerås, Sweden

August 2018
Abstract

Process-based arguments aim at demonstrating that a process, compliant with a standard, has been followed during the development of a safety-critical system. Compliance with these processes is mandatory for certification purposes, so the generation of process-based arguments is essential, but also a very costly and time-consuming task. In addition, inappropriate reasoning in the argumentation, such as insufficient evidence (i.e., a fallacious argumentation), may result in a loss of quality of the system, leading to safety-related failures. Therefore, avoiding or detecting fallacies in process-based arguments is crucial. However, the process of reviewing such arguments is currently done manually and is based on the expert’s knowledge, so it is a very laborious and error-prone task.

In this thesis, an approach to automatically generate fallacy-free process-based arguments is proposed and implemented. This solution is composed of two parts: (i) detecting omission of key evidence fallacies on the modelled processes, and (ii) transforming them into process-based safety arguments. The former checks automatically if the process model, compliant with the Software & Systems Process Engineering Metamodel (SPEM) 2.0, contains the sufficient information for not committing an omission of key evidence fallacy. If fallacies are detected, the functionality provides the proper recommendation to resolve them. Once the safety engineers/process engineers modify the process model following the provided recommendations, the second part of the solution can be applied. This one generates automatically the process-based argument, compliant with the Structured Assurance Case Metamodel (SACM), and displays it—rendered via Goal Structuring Notation (GSN)—into the OpenCert assurance case editor within the AMASS platform. The applicability of the solution is validated in the context of the ECSS-E-ST-40C standard.
Table of Contents

1. Introduction ...................................................................................................................... 9
   1.1. Motivation .................................................................................................................. 9
   1.2. Context ....................................................................................................................... 9
   1.3. Contribution .............................................................................................................. 9
   1.4. Thesis structure ........................................................................................................ 10

2. Background ..................................................................................................................... 11
   2.1. Process and SPEM 2.0-based process modelling ..................................................... 11
   2.2. Safety arguments and modelling ............................................................................. 14
       2.2.1. Text-based notations ....................................................................................... 15
       2.2.2. Graphics-based notations ............................................................................. 16
   2.3. Argumentation fallacies ......................................................................................... 20
   2.4. Model-driven engineering ....................................................................................... 21
       2.4.1. Model-driven architecture ............................................................................. 22
       2.4.2. Model-driven development .......................................................................... 23
       2.4.3. Model transformations .................................................................................. 23

3. Related work ................................................................................................................... 25

4. Problem formulation and analysis ................................................................................. 27
   4.1. Problem formulation ............................................................................................... 27
   4.2. Problem analysis ...................................................................................................... 27
       4.2.1. Detect the omission of key evidence fallacy problem .................................... 27
       4.2.2. Transform the validated process into a safety argumentation problem ............ 28

5. Method ............................................................................................................................. 29
   5.1. Research methodology ............................................................................................ 29
   5.2. Methods to solve the problem ................................................................................. 29
       5.2.1. Detect the omission of key evidence fallacy method ...................................... 30
       5.2.2. Transform the validated process into a safety argumentation method .......... 34

6. Solution ............................................................................................................................ 37
   6.1. Detect the omission of key evidence fallacy solution ............................................. 37
       6.1.1. Java implementation ....................................................................................... 38
       6.1.2. Fallacy detection plugin ................................................................................. 42
   6.2. Transform the validated process into a safety argumentation solution ................. 43
       6.2.1. Model to model transformation .................................................................... 44
       6.2.2. Transformation plugin .................................................................................. 48

7. Case study ....................................................................................................................... 53
   7.1. Description ............................................................................................................... 53
   7.2. Process modelling in EPF Composer ..................................................................... 55
   7.3. Omission of key evidence fallacy detection application ....................................... 58
   7.4. Transformation application .................................................................................... 60
   7.5. Discussion ............................................................................................................... 62
8. **Conclusions** ............................................................................................................. 63
   8.1. Summary ................................................................................................................. 63
   8.2. Future work ............................................................................................................. 63

References ...................................................................................................................... 65

Appendix .......................................................................................................................... 69
List of Figures

Figure 1. Standardised architecture................................................................................................................. 11
Figure 2. Relationship between Task, Role and Work Product elements [10]............................................................ 12
Figure 3. SPEM 2.0 Method Framework [8]........................................................................................................... 12
Figure 4. Safety-process modelling with SPEM 2.0 [11].............................................................................................. 12
Figure 5. Essential parts of a safety case and their relationship [13]......................................................................... 14
Figure 6. Example of Normal Prose argument [16].................................................................................................... 15
Figure 7. Example of Structured Prose argument [16].............................................................................................. 15
Figure 8. Example of Argument Outline [16]........................................................................................................... 15
Figure 9. Example of Mathematical Proof Style argument [16].............................................................................. 16
Figure 10. Example of LISP Style argument [16]...................................................................................................... 16
Figure 11. Example of Goal element ...................................................................................................................... 17
Figure 12. Example of Strategy element ................................................................................................................ 17
Figure 13. Example of Context element ................................................................................................................ 17
Figure 14. Example of Solution element ................................................................................................................ 18
Figure 15. Example of Justification element .......................................................................................................... 18
Figure 16. Example of Assumption element template ............................................................................................ 18
Figure 17. Example of Undeveloped Goal ........................................................................................................... 18
Figure 18. SupportedBy relationship .................................................................................................................... 19
Figure 19. InContextOf relationship ..................................................................................................................... 19
Figure 20. CAE example [19]................................................................................................................................. 19
Figure 21. Relationship between MDE, MDA, and MDE [23]................................................................................... 22
Figure 22. Four-layered Metamodel Architecture adapted from [27]..................................................................... 22
Figure 23. Overview of MDA [26].......................................................................................................................... 23
Figure 24. Research methodology adapted from [45]............................................................................................. 29
Figure 25. UMA metamodel of the Method Content [46]........................................................................................ 30
Figure 26. UMA metamodel of the Process elements [46]..................................................................................... 31
Figure 27. Practice customization with icon .......................................................................................................... 32
Figure 28. Standard's requirements capture .......................................................................................................... 32
Figure 29. Modelled safety process....................................................................................................................... 32
Figure 30. Standard's requirements mapping ...................................................................................................... 33
Figure 31. Activity diagram for fallacy detection method ..................................................................................... 33
Figure 32. Structure of the Argumentation Metamodel [50]................................................................................... 35
Figure 33. Activity diagram for the transformation method .................................................................................. 36
Figure 34. Flowchart of fallacy detection process ................................................................................................. 37
Figure 35. Extraction of standard's requirements implementation ...................................................................... 38
Figure 36. Get selected process element implementation .................................................................................... 39
Figure 37. Get process elements implementation ................................................................................................ 39
Figure 38. Get tool qualifications implementation ................................................................................................ 40
Figure 39. Remove additional information implementation .................................................................................. 40
Figure 40. List of prepositions and articles ........................................................................................................... 41
Figure 41. Fallacy decision implementation ........................................................................................................ 41
Figure 42. Method to store the omitted evidence ................................................................................................. 42
Figure 43. Fallacy detection plugin menu ............................................................................................................. 42
Figure 44. Method to select the target directory of the report ................................................................................ 43
Figure 45. Validation completion dialog ............................................................................................................ 43
Figure 46. Get skills operation ............................................................................................................................. 44
Figure 47. Flowchart of transformation plugin .................................................................................................... 45
Figure 48. Evidence transformation .................................................................................................................... 46
Figure 49. Get tool qualifications operation ........................................................................................................ 47
Figure 50. CapabilityPattern2Claim rule snippet ............................................................................................... 47
Figure 51. Context transformation ....................................................................................................................... 48
Figure 52. Model repository structure .................................................................................................................. 48
Figure 53. Obtaining the models .......................................................................................................................... 49
Figure 54. Creation of EMF models ..................................................................................................................... 50
Figure 55. Connection to the CDO Repository ..................................................................................................... 50
Figure 56. Load model into CDO Repository ........................................................................................................ 51
List of Tables

Table 1. Key Method Content elements ................................................................. 13
Table 2. Key Process elements ........................................................................... 13
Table 3. Mapping between SACM, GSN and CAE ............................................. 20
Table 4: Taxonomy of Fallacies [21] ................................................................. 21
Table 5. Mapping elements ................................................................................ 34
Table 6. Roles requirements ................................................................................ 54
Table 7. Evidence added after validation ............................................................ 59
# Acronyms and Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMASS</td>
<td>Architecture-driven, Multi-concern and Seamless Assurance and Certification of Cyber-Physical Systems</td>
</tr>
<tr>
<td>AOCS</td>
<td>Attitude and Orbit Control Subsystem</td>
</tr>
<tr>
<td>CACM</td>
<td>Common Assurance and Certification Metamodel</td>
</tr>
<tr>
<td>CAE</td>
<td>Claims-Arguments-Evidence</td>
</tr>
<tr>
<td>CDO</td>
<td>Connected Data Objects</td>
</tr>
<tr>
<td>CPS</td>
<td>Cyber-Physical Systems</td>
</tr>
<tr>
<td>ECSS</td>
<td>European Cooperation for Space Standardization</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modelling Framework</td>
</tr>
<tr>
<td>EN</td>
<td>European Norms</td>
</tr>
<tr>
<td>EPF</td>
<td>Eclipse Process Framework</td>
</tr>
<tr>
<td>ETL</td>
<td>Epsilon Transformation Language</td>
</tr>
<tr>
<td>GSN</td>
<td>Goal Structuring Notation</td>
</tr>
<tr>
<td>HAZOP</td>
<td>Hazard and Operability</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
</tr>
<tr>
<td>M2M</td>
<td>Model-to-Model</td>
</tr>
<tr>
<td>MDA</td>
<td>Model-Driven Architecture</td>
</tr>
<tr>
<td>MDD</td>
<td>Model-Driven Development</td>
</tr>
<tr>
<td>MDE</td>
<td>Model-Driven Engineering</td>
</tr>
<tr>
<td>MDSafeCer</td>
<td>Model-Driven Safety Certification</td>
</tr>
<tr>
<td>MOF</td>
<td>MetaObject Facility</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>PML</td>
<td>Process Modelling Language</td>
</tr>
<tr>
<td>RMC</td>
<td>Rational Method Composer</td>
</tr>
<tr>
<td>RUP</td>
<td>Rational Unified Process</td>
</tr>
<tr>
<td>SACM</td>
<td>Structured Assurance Case Metamodel</td>
</tr>
<tr>
<td>SPEM</td>
<td>Software &amp; System Process Engineering Metamodel</td>
</tr>
<tr>
<td>SW</td>
<td>Software</td>
</tr>
<tr>
<td>UMA</td>
<td>Unified Method Architecture</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>V&amp;V</td>
<td>Verification and Validation</td>
</tr>
<tr>
<td>XMI</td>
<td>XML Metadata Interchange</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
<tr>
<td>XP</td>
<td>eXtreme Programming</td>
</tr>
</tbody>
</table>
1. Introduction

This chapter presents a brief introduction to the work done in this thesis. Section 1.1 explains the main motivation for this work, and the context in which the thesis is developed is explained in Section 1.2. Section 1.3 highlights the main contributions of this thesis. Finally, Section 1.4 gives the idea about the main structure of the thesis.

1.1. Motivation

Standards such as ECSS-E-ST-40C for space [6], ISO 26262 for automotive [3], and EN 50128 for railway [5] provide guidance on the steps and processes that shall be followed during the development of a safety-critical system. Compliance with these normative processes is mandatory for certification of such safety-critical systems. In this context, a safety case, which is a contextualised structured argument, shows that a system is acceptably safe. To accomplish this, the process-based arguments aim to show that the process used to develop the system contributes to its safety, that is, argue about the safety-related decisions and explain how the presented evidence relating and contributing to the safety goals.

However, the creation of process-based arguments is a very costly, time-consuming and laborious task, because inappropriate reasoning in the argumentation through irrelevant premises or insufficient evidence, i.e. a fallacious safety argument, can undermine the credibility of the argument, as well as deteriorate the quality and safety of the system, leading to safety-related failures [14], [20]. Therefore, the avoidance, detection, and removal of fallacies in the argumentation is an essential task to reduce the risk of failures in a safety-critical system. However, since this task depends on the experience and knowledge of the developer, it is also prone to errors.

In order to solve these issues, a model-driven approach can be used to automatically generate process-based arguments from process models [11] and incorporating assistance to avoid these fallacies when engineers are creating safety arguments.

This thesis aims to provide such assistance for safety engineers by developing an automatic method of detecting existing fallacies in the modelled process. Once these fallacies are detected and solved, the safety argumentation is generated automatically from the process model. To reach these goals, this thesis tries to answer the following research questions:

- How do we detect the omission of key evidence fallacy in process-based arguments?
- Can we provide process compliance with argumentation?
- Can we increase the quality of safety arguments?

1.2. Context

This thesis is defined in the scope of the ongoing research project AMASS (Architecture-driven, Multi-concern and Seamless Assurance and Certification of Cyber-Physical Systems), which “will create and consolidate the de-facto European-open wide open tool platform, ecosystem, and self-sustainable community for assurance and certification of Cyber-Physical Systems (CPS) in the largest industrial vertical markets including automotive, railway, aerospace, space, energy” [42].

The platform proposed by AMASS project integrates several tools for deriving assurance certification elements and is built on top of several Eclipse and PolarSys projects. Since we aim at integrating the solution in this platform, we have to adapt to the existing constraints, so the current process modelling tool (i.e., EPF Composer [46]) and assurance case editor (i.e., OpenCert [43]) are used to implement the functionalities.

1.3. Contribution

This thesis will contribute to enabling the generation of fallacy-free process-based safety argumentations from process models. For this, a Fallacy Detection plugin will be developed within the AMASS platform, which allows safety/process engineers to validate whether the process models contain sufficient information, preventing thereby the occurrence of fallacy (i.e., omission of key evidence) in process-based argumentations. Through this thesis, the following outcomes are provided:

- Study state of the art of safety cases and fallacy detection mechanisms, as well as the automatic generation of safety argumentation.
• The design and implementation of an algorithm to automatically detect fallacies (i.e., omission of key evidence) in model-based safety-processes compliant with the OMG standard for modelling processes SPEM 2.0, moreover an analysis of the metamodels elements involved in it.

• Enhance existing implementation: the activation of process-based arguments generator plugin from the EPF Composer, incorporating the presented –and validated– evidence as solutions or justifications.

• An illustration of the applicability of the solution through a case study in the context of the space domain (i.e., compliant with ECSS-E-40C standard).

In addition, as an early result of this thesis, a scientific article [57], which includes the concept and design of this approach, has been presented and accepted in the 11th International Conference on the Quality of Information and Communications Technology (QUATIC).

1.4. Thesis structure

The rest of the report is structured as follows:

Chapter 2 presents the background knowledge needed to understand the rest of the report.

Chapter 3 presents the state of the art of the assurance, validation and generation of safety arguments through an analysis of the related work.

Chapter 4 presents the problem formulation and analysis, which determines the rest of the work. In addition to the raised research questions, the goal of this thesis is divided into subproblems, which allows addressing these questions and find a proper solution.

Chapter 5 presents the scientific methodology used in this thesis, which is based on analysing a problem, finding a solution to the problem, and evaluating the solution.

Chapter 6 presents the solution proposed by this thesis. This solution is composed of two different plugins that perform the fallacy detection functionality, and the transformation process to automatically generate the safety argumentation from the process model.

Chapter 7 introduces a case study to illustrate the applicability of the proposed solution. The modelled process represents the planning phase of the AOCS software development process, following Clause 5.5 of the ECSS-E-ST-40C standard.

Chapter 8 presents the conclusions obtained from the results of the thesis, including the limitations found during its development, and some lines of future work to improve it.
2. Background

This chapter explains the needed background concepts to be able to understand the rest of the document. For this, Section 2.1 presents the process models and process modelling language, specifically SPEM 2.0. Section 2.2 describes the basic notions of a safety case, such as text-based notations (Section 2.2.1) and graphic-based notations (Section 2.2.2). Section 2.3 presents the argumentation fallacies, and finally, Section 2.4 presents the main concepts of the model-driven engineering approaches.

2.1. Process and SPEM 2.0-based process modelling

A process can be defined as “a series of actions or steps taken in order to achieve a particular end” [1]. More specifically, in the engineering domain, a process is “a series of interrelated tasks that, together, transform inputs into outputs” [2]. The growing complexity, the large number of people involved in the development of software products, as well as the wide range of information that these development teams need and use, has increased the necessity to follow processes to generate quality products and facilitate the development of them.

However, there are systems whose quality and safety must be thoroughly reviewed, called safety-critical systems. These systems are defined as those whose failure can cause loss of life, damage to property or environment. Due to this, numerous standards in the field of safety-critical systems such as ISO 26262 [3] (automotive), DO-178C [4] (airplane), EN 50128 [5] (railway), or ECSS-E-ST-40C [6] (space), specify the activities and tasks –and their order– that must be carried out during the development phase of a safety-critical system. These safety processes are also known as safety-life cycles, and their fulfilment is needed for certification purposes. In this way, they are used to ensure the quality of the developed systems, and thus, to argue that the developed system is safe.

SPEM (Software & System Process Engineering Metamodel), an Object Management Group’s (OMG) standard is one of the Process Modelling Languages (PMLs) that describes processes based on MOF (MetaObject Facility) [7]. Figure 1 shows the four-level standardised architecture proposed by OMG, where the upper level (M3) is the meta-metamodel MOF, and the immediately lower level (M2) is the metamodel SPEM. The lower layer (M1) represents the process model such as RUP, Scrum or XP. Finally, the lowest level (M0) represents the actual process. Since these concepts are related to MDE (Model-Driven Engineering), they will be further explained in Section 2.4.

<table>
<thead>
<tr>
<th>M3: MetaObject Facility</th>
<th>MOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>M2: Process Metamodel</td>
<td>SPEM</td>
</tr>
<tr>
<td>M1: Process Model</td>
<td>e.g. RUP, SCRUM, XP</td>
</tr>
<tr>
<td>M0: Performing Process</td>
<td>Software project</td>
</tr>
</tbody>
</table>

Figure 1. Standardised architecture

Specifically, SPEM is “a process engineering meta-model as well as conceptual framework, which can provide the necessary concepts for modelling, documenting, presenting, managing, interchanging, and enacting development methods and processes” [8]. The first version was introduced in 2002 and was built upon UML 1.4. In 2005 appeared the second version, SPEM 1.1 [9], incorporating minor updates and finally, in 2008, the current version SPEM 2.0 was released, which is compliant with UML 2. This includes significant benefits such as greatly improved modelling techniques, graphic interchange capability, and a more modular organisation [8]. In this way, SPEM 2.0 leverages these functionalities and the ability to work with UML 2 tools, enhancing its process modelling techniques and capabilities.

SPEM is based on the idea that a process can be defined as a collaboration between different entities. In particular, SPEM 2.0, through the Method Content package, allows defining a process through the main elements Task, Role and Work Product, where tasks can be grouped into larger structures such as activities and phases (work definition elements). Figure 2 shows the relationship between these elements, where a Role is responsible for one or more Work Product(s), and one or more Role(s) performs one or more Task(s). These Tasks have one or more Work Product(s) as inputs and outputs.
In addition, *Guidance* and *Tool* elements can be used to represent the resources and tools that should be used, respectively. This package defines the “what, who and how” of the work that must be done in a process, while the “when” is represented by the *Process* package. Figure 3 shows the main elements of both packages. While the *Method content* package provides the concepts to define static and reusable process elements, that is, independent of any specific process or development project, the *Process* package places these concepts into the context of a specific life-cycle. The key *Method Content* elements (left-hand side of the diagram) are further described in Table 1. An example of a safety process modelling with SPEM 2.0 is shown in Figure 4.

![Figure 2. Relationship between Task, Role and Work Product elements [10]](image)

In addition, *Guidance* and *Tool* elements can be used to represent the resources and tools that should be used, respectively. This package defines the “what, who and how” of the work that must be done in a process, while the “when” is represented by the *Process* package. Figure 3 shows the main elements of both packages. While the *Method content* package provides the concepts to define static and reusable process elements, that is, independent of any specific process or development project, the *Process* package places these concepts into the context of a specific life-cycle. The key *Method Content* elements (left-hand side of the diagram) are further described in Table 1. An example of a safety process modelling with SPEM 2.0 is shown in Figure 4.

![Figure 3. SPEM 2.0 Method Framework [8]](image)

On the right-hand side of the diagram are the SPEM 2.0 *Process* elements. As said before, the *Method Content* elements can be placed and applied in a specific life-cycle process. When that happens, reference classes to the static *Method Content* classes are created, referred to as *Method Content Use* (i.e., *Task Use*, *Role Use*, and *Work Product Use*). These classes referred to their *Method Content* ones, but can store individual changes with respect to them. In addition to these referred classes, the main *Process* element is the *Activity*, which allows defining *Breakdown* structures and relationships to define a workflow. Table 2 includes a brief description of the main *Process* elements.

![Figure 4. Safety-process modelling with SPEM 2.0 [11]](image)
Laura Gómez Rodríguez  
A tool-supported method for fallacies detection in process-based argumentation

Table 1. Key Method Content elements

<table>
<thead>
<tr>
<th>Element</th>
<th>Icon</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task</td>
<td><img src="image1" alt="Icon" /></td>
<td>Describes an assignable unit of work. This work has to be performed by one or more Role(s) and has input and output Work Products. A Task can be defined by different steps.</td>
<td>Define requirements, Define Use Cases, Identify Hazards</td>
</tr>
<tr>
<td>Role</td>
<td><img src="image2" alt="Icon" /></td>
<td>Describes who is responsible for the work. A Role can be an individual (e.g. project manager) or a set of individuals (e.g. development team). It also defines related skills, competencies, and responsibilities. A Role may be responsible for one or more Work Product(s) and may perform one or more Task(s).</td>
<td>Developer, designer, project manager, system analyst, software engineer</td>
</tr>
<tr>
<td>WorkProduct</td>
<td><img src="image3" alt="Icon" /></td>
<td>Defines tangible artefacts consumed, produced, or modified by a Task. A Task has Work Products as input and output. A Work Product can be optional or mandatory. The types of work products are: Artifacts, Deliverables, and Outcomes.</td>
<td>Functional requirements, HAZOP analysis</td>
</tr>
<tr>
<td>Tool</td>
<td><img src="image4" alt="Icon" /></td>
<td>Describes the capabilities of a tool to perform the work defined in a Task. It can be defined as useful, recommended or necessary for a Task's completion.</td>
<td>Use Cases tool, HAZOP package</td>
</tr>
<tr>
<td>Guidance</td>
<td><img src="image5" alt="Icon" /></td>
<td>Describes additional information related to any model element to, for example, define how to perform the work defined in a Task. The specific type of Guidance must be selected. The types of guidance are: Checklists, Concepts, Examples, Guidelines, Estimation Considerations, Practices, Reports, Reusable Assets, Roadmaps, Supporting Materials, Templates, Term Definitions, Tool Mentors, and Whitepapers.</td>
<td>Unified Modelling Language (UML), HAZOP package and guidelines</td>
</tr>
</tbody>
</table>

Table 2. Key Process elements

<table>
<thead>
<tr>
<th>Element</th>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TaskUse</td>
<td><img src="image6" alt="Icon" /></td>
<td>Represents a Task in the context of a specific activity. One Task can be represented by many Task Uses, each with a different set of relationships (with Role Uses and Work Product Uses) depending on the activity in which is instantiated. In addition, the Task Use element indicates what subset of steps shall be performed at that particular point in the process.</td>
</tr>
<tr>
<td>RoleUse</td>
<td><img src="image7" alt="Icon" /></td>
<td>Represents a Role in the context of a specific activity. One Role can be represented by many Role Uses, each with different relationships and responsibilities depending on the specific activity in which it is involved.</td>
</tr>
<tr>
<td>WorkProduct Use</td>
<td><img src="image8" alt="Icon" /></td>
<td>Represents a Work Product in the context of a specific activity. One Work Product can be represented by many Work Product Uses, where each one can include different relationships depending on the specific activity in which it is involved.</td>
</tr>
<tr>
<td>Process</td>
<td><img src="image9" alt="Icon" /></td>
<td>Describes the structure for particular types of development projects or part of them, by adapting to the specific situation and needs of the project, and assigning the proper method and process elements.</td>
</tr>
<tr>
<td>Activity</td>
<td><img src="image10" alt="Icon" /></td>
<td>Represents a set of nested Breakdown Elements, that is, any type of Process element that is part of a breakdown structure. This can include other Activity instances such as Phases, Task Uses, etc.</td>
</tr>
<tr>
<td>Phase</td>
<td><img src="image11" alt="Icon" /></td>
<td>Represents a significant period in a project, which usually ends with a milestone or a set of deliverables. It is a special predefined Activity since its use has a great significance in the definition of breakdown structures. A Phase can be considered as an Activity which is not repeatable.</td>
</tr>
<tr>
<td>Delivery Process</td>
<td><img src="image12" alt="Icon" /></td>
<td>Describes a complete approach, i.e., that covers the development life-cycle from beginning to end, for performing a specific project type. It provides a complete life-cycle model with predefined phases and activities, detailed by arranging the referred method content in breakdown structures. In addition, it defines what produces and by who, in the form of Work Product Usage and Team Allocation structures.</td>
</tr>
<tr>
<td>Capability Pattern</td>
<td><img src="image13" alt="Icon" /></td>
<td>Also called Process Pattern, describes a reusable cluster of Activities that provide a consistent approach to common structures, such as “use case-based requirements management” or “develop components”. It does not relate to any specific phase of a development life-cycle.</td>
</tr>
</tbody>
</table>
2.2. Safety arguments and modelling

Safety cases are gaining importance in different areas such as automotive, railway, aerospace or nuclear since they are mandatory for certification purposes. Due to the growing number of fields, organisations and people that work with safety-critical systems, different definitions for safety cases can be found. According to the U.K. Ministry of Defence, a safety case is “a structured argument, supported by a body of evidence that provides a compelling, comprehensible and valid case that a system is safe for a given application in a given operating environment” [12]. This definition, therefore, describes the general structure of a safety case, shown in Figure 5.

![Figure 5. Essential parts of a safety case and their relationship [13]](image)

The previous figure shows that the role of the safety argument is, therefore, the relationship between the evidence and the objectives. According to Kelly, “a safety case should communicate a clear, comprehensive and defensible argument that a system is acceptably safe to operate in a particular context” [13]. That is, not only substantial supporting evidence is necessary for a safety case to be successful, but it is also essential to explain clearly and convincingly how this evidence relating to the safety objectives. Therefore, both parties are essential since a safety case without evidence is unfounded, and a safety case without argument is unexplained.

A safety argument is therefore the part of a safety case that tries to explain and relate how the presented evidence allows ensuring that the system complies with the safety requirements and objectives demanded by all the stakeholders involved. More formally, a safety argument, like any other argument, is a claim supported by other claims or premises. Damer defines in [14] that “an argument is constituted by two or more explicit and / or implicit claims, one or more of which supports or provides evidence for the truth or merit of another claim, the conclusion.”

Arguments can be classified as deductive or inductive arguments. The former is one in which the premises guarantee the truth of the conclusion, which is a logical consequence of the premises; that is, if the premises are true, then the conclusion must also be true. The latter is the one where the conclusion is taken as true, and the premises provide reasons to support the alleged veracity of the conclusion. However, in this type of arguments the conclusion does not follow any logical consequence with respect to the premises, so the truth of all its premises would not imply the truth of the conclusion. Safety arguments are of this type, i.e., inductive arguments.

Safety arguments can also be classified as product-based or process-based. The first ones are focused on ensuring the quality and safety of the attributes and objectives of the finished product. The second ones are focused on the quality and adequacy of the process followed during the development of the product. However, although these types of argumentation can be represented individually, they can also be related, and an argument can include process and product-based sub-arguments. As stated in the previous section, following a process improves the quality of the element developed, therefore, process-based arguments can ensure that the product-based evidence is trustworthy since it has followed the proper process.

To document safety arguments within a safety case, several approaches exist both textual (e.g., normal prose, structured prose, argument outline, mathematical proof style, LISP style) and graphical (e.g., Goal Structuring Notation, Claims-Arguments-Evidence) [15], which are presented below. Besides, the Object Management Group offers a standardised modelling language for representing safety cases; the Structured Assurance Case Metamodel (SACM) 2.0 [16]. It aims to unify and standardise the graphical notations broadly used for documenting safety cases, namely GSN and CAE.
## 2.2.1. Text-based notations

- **Normal prose**

  Commonly used in disciplines such as law or philosophy, safety cases can be written using natural language. Figure 6 shows a fragment of a safety argument written in normal prose.

<table>
<thead>
<tr>
<th>The control system is acceptably safe, given a definition of acceptably safe, because all identified hazards have been eliminated or sufficiently mitigated and the software has been developed to the integrity levels appropriate to the hazards involved.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given both the tolerability targets for hazards (from reference Z), and the list of hazards identified from the functional hazard analysis (from reference Y), we can show that all identified hazards have been identified or sufficiently mitigated by arguing over all three of the identified hazards: H1, H2, and H3.</td>
</tr>
<tr>
<td>We know from the formal verification we conducted that H1 has been eliminated.</td>
</tr>
<tr>
<td>We know that catastrophic hazard H2 has been sufficiently mitigated because fault tree analysis [...].</td>
</tr>
<tr>
<td>We know that major hazard H3 has been sufficiently mitigated because fault tree analysis [...].</td>
</tr>
</tbody>
</table>

**Figure 6. Example of Normal Prose argument [16]**

- **Structured prose**

  One of the main problems of the normal prose approach is the lack of structure in the argument, as well as the difficulty to identify the main elements of the argument (e.g., claims and strategies). With the structured prose approach, it is intended to improve these problems with constrained prose to denote these essential parts explicitly. Figure 7 shows an example of a structured prose argument fragment.

<table>
<thead>
<tr>
<th>This argument establishes the following claim: the control system is acceptably safe, within the context of a definition of acceptably safe. To establish the top-level claim, two sub-claims are established: (1) all identified hazards have been eliminated or sufficiently mitigated and (2) the software has been developed to the integrity levels appropriate to the hazards involved.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Within the context of the tolerability targets for hazards (from reference Z) and the list of hazards identified from the functional hazard analysis (from reference Y), we follow the strategy of arguing over all three of the identified hazards (H1, H2, and H3) to establish sub-claim 1, yielding three additional claims: H1 has been eliminated; H2 has been sufficiently mitigated; and H3 has been sufficiently mitigated.</td>
</tr>
<tr>
<td>The evidence that H1 has been eliminated is formal verification.</td>
</tr>
<tr>
<td>The evidence that catastrophic hazard H2 has been sufficiently mitigated is a fault tree analysis [...].</td>
</tr>
<tr>
<td>The evidence that the major hazard H3 has been sufficiently mitigated is a fault tree analysis [...].</td>
</tr>
</tbody>
</table>

**Figure 7. Example of Structured Prose argument [16]**

- **Argument outline**

  This approach allows defining the structure of the argument even more explicitly. To do this, simple phrases are used to define each of the elements with a different type of font (bold - claim, italic - context, etc.), as well as indentation and numbering to highlight the hierarchy of the elements. Figure 8 shows an example of this approach.

<table>
<thead>
<tr>
<th>Claim 1: Control system is acceptably safe.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Context 1: Definition of acceptably safe.</strong></td>
</tr>
<tr>
<td><strong>Claim 1.1: All identified hazards have been eliminated or sufficiently mitigated.</strong></td>
</tr>
<tr>
<td><strong>Context 1.1-a: Tolerability targets for hazards (reference Z).</strong></td>
</tr>
<tr>
<td><strong>Context 1.1-b: Hazards identified from functional hazard analysis (reference Y).</strong></td>
</tr>
<tr>
<td><strong>Strategy 1.1: Argument over all identified hazards (H1, H2, H3).</strong></td>
</tr>
<tr>
<td><strong>Claim 1.1.1: H1 has been eliminated.</strong></td>
</tr>
<tr>
<td>Evidence 1.1.1: Formal verification.</td>
</tr>
<tr>
<td><strong>Claim 1.1.2: Probability of H2 occurring &lt; 1x10^-6 per annum.</strong></td>
</tr>
<tr>
<td>Justification 1.1.2: 1x10^-6 per annum limit for catastrophic hazards.</td>
</tr>
<tr>
<td>Evidence 1.1.2: Fault Tree analysis.</td>
</tr>
<tr>
<td><strong>Claim 1.1.3: Probability of H3 occurring &lt; 1x10^-3 per annum.</strong></td>
</tr>
<tr>
<td>Justification 1.1.3: 1x10^-3 per annum limit for major hazards.</td>
</tr>
<tr>
<td>Evidence 1.1.3: Fault tree analysis.</td>
</tr>
</tbody>
</table>

**Figure 8. Example of Argument Outline [16]**
Mathematical proof style

The mathematical proof approach allows structuring the argument through claims (defined with the word “Establish”), followed by its context (under the label “Given”), and a series of statements and reasons organised in a table as Figure 9 shows. This type of approach allows making references to established claims later, so it maintains the top-down nature of the argumentation [17].

| Establish: SystemSafe: Control system is acceptably safe. |
| Given: A. Definition of acceptably safe. |
| Statements | Reasons |
| 1. All identified hazards have been eliminated or sufficiently mitigated | 1. HazardsHandled (established below) |
| 2. The software has been developed to the integrity level appropriate to the hazards identified | 2. ProcessAcceptable (established below) |
| 3. Control system is acceptably safe | 3. 1, 2 |

| Establish: HazardsHandled: All identified hazards have been eliminated or sufficiently mitigated. |
| Given: A. Tolerability targets for hazards (reference Z). B. Hazards identified from functional hazard analysis (reference Y). |
| By: Arguing over all identified hazards (H1, H2, H3) |
| Statements | Reasons |
| 1. H1 has been eliminated | 1. formal verification |
| 2. p(H2) < 1x10^{-6} per annum (p.a.) | 2. fault tree analysis |
| 3. Upper limit on permitted catastrophic hazard occurrence is 1x10^{-6} p.a. | 3. Given (A) |
| 4. H2 has been mitigated | 4. 2, 3 |
| ... | ... |

Figure 9. Example of Mathematical Proof Style argument [16]

LISP Style

This approach is based on the LISP programming language. Each element of the argument appears at the beginning of a list, indexed in such a way that it applies to everything that follows within that list. Figure 10 shows an extract of an example of this type of notation.

```lisp
(context DefSafe (claim SystemSafe (context FHAHazards (context Targets (claim HazardsHandled (strategy ArgOveHaz (claim H1Elim (evidence H1Evidence)) (claim H2OK (justification CatHaz) (evidence FTA)))))
```

Figure 10. Example of LISP Style argument [16]

2.2.2. Graphics-based notations

As mentioned above, besides the already presented text-based notations there are other methods to represent safety arguments, such as graphics-based ones namely GSN and CAE, which are described below.
Goal Structuring Notation (GSN)

GSN is one of the most commonly used notations to represent safety arguments. It can be defined as “a graphical argument notation which can be used to document explicitly the elements and structure of an argument and the argument’s relationship to evidence” [18]. It is based on the Toulmin’s argumentation theory called “Toulmin Model” and on goal-based approaches to requirements engineering, such as KAOS. The main characteristic of this notation is the organisation of the argument in ‘goal structures’. A goal structure shows how the goals of a system are broken down in turn into different sub-goals until reaching those in which direct evidence is sufficient. Therefore, a goal structure is formed by the different types of elements existing in a safety argument and the relationship between them. The symbols and concrete syntax to describe these elements are described below.

Core elements

The main elements in GSN, also known as nodes, are the following:

- **Goal**
  As stated in Section 2.2, an argument is composed of a series of premises that supports the truth of the main premise (e.g., the safety of a system). Each of these premises or claims of a system—or of a property of the system—are represented in GSN as goals. This element is rendered as a rectangle. An example of a goal can be seen in Figure 11.

- **Strategy**
  When a goal is broken-down into sub-goals, a strategy is used to explain what reasoning has been followed to arrive at that decomposition, that is, how or why a goal is decomposed into sub-goals. Therefore, it is used to provide additional description or explanation of the reasoning about a connection or relationship between one or more Goals (premises) to another Goal (conclusion). The symbol used is the parallelogram, as can be seen in Figure 12.

- **Context**
  The context element defines the boundaries in which a goal or strategy should be interpreted. This element thus allows providing additional information to support the core reasoning of the argument. It is represented by a rectangle with rounded corners as shown in Figure 13.

- **Solution**
  As stated before, a ‘goal structure’ ends when a goal can be solved by direct evidence instead of by decomposition into sub-goals. Direct evidence is represented in GSN as solutions and rendered as a circle as shown in Figure 14. A solution is, therefore, a reference to an evidence item, and it usually is an analysis result (e.g., FMECA, HAZOP, FTA), a verification tests result, a certification, etc.
• **Justification**

This element provides additional reasoning to support another element, for example, the choice of a specific strategy. It is also used to provide additional description or rationales about a premise, which is intended not supported by evidence. A Justification element is rendered as an oval with a letter ‘J’ at the bottom-right as shown in Figure 15.

![Justification Element](image)

**Figure 15. Example of Justification element**

• **Assumption**

This element presents an intentionally unfounded statement, which must be taken as true without providing further explanations. It is rendered as an oval with a letter ‘A’ at the bottom-right as shown in Figure 16.

![Assumption Element](image)

**Figure 16. Example of Assumption element**

• **Undeveloped entity**

A hollow diamond at the centre-bottom of a goal indicates that the element has been intentionally left undeveloped in the argument. Figure 17 shows an example of this element.

![Undeveloped Goal Element](image)

**Figure 17. Example of Undeveloped Goal**

**Relationships**

The elements presented above are combined to represent goal structures, creating thus the safety argument. This is done in GSN by linking them, based on their relationship, using one of the following types of connectors:

• **SupportedBy**

Rendered as a line with solid arrowhead, it is used both for inferential relationships; i.e., those that show the relationships between different goals, and for evidential relationships; i.e., those that show the relationship between different goals, and for evidential relationships; i.e., those that show the relationship between a goal and its supporting evidence. Figure 18 shows a representation of this relationship.
The permitted supportedBy connections are: goal-to-goal, goal-to-strategy, goal-to-solution, strategy-to-goal.

Figure 18. SupportedBy relationship

- InContextOf

Rendered as a line with hollow arrowhead, it is used to contextual relationships. Figure 19 shows the representation used for this kind of relationship.

The permitted inContextOf connections are: goal-to-context, goal-to-assumption, goal-to-justification, strategy-to-context, strategy-to-assumption, and strategy-to-justification.

Figure 19. InContextOf relationship

Syntax

As Figures 11 to 17 show, all the core GSN elements are represented by a graphical symbol and a textual statement. In addition, an optional identifier can be added to each element, represented here in brackets. If identifiers are included, they must uniquely identify each element. Regarding the textual content of the nodes, there is no formal syntax to describe it. However, informally the goal and justification elements shall be a noun phrase plus a verb phrase sentence, while the solution elements shall be a noun-phrase.

➢ Claim-Argument-Evidence (CAE)

Besides GSN, there is another graphical as well as textual notation called Claim-Argument-Evidence (CAE) [19]. It is based on the same principle as GSN, but with a different representation of the main elements. These elements are the followings:

- **Claim**: rendered as a blue ellipse, it is a statement within an argument that can be true or false. It is supported by sub-claims, arguments, or evidence, and can also be defined by additional information as context.

- **Argument**: rendered as a green rounded rectangle, it is an optional element used to provide a description of the approach of the argument.

- **Evidence**: rendered as a magenta rectangle, it is a reference to the supporting evidence of the claim.

In addition to these key elements, the relationship between them is modelled through a coloured arrow, depending on the colour of the source element, which in turn can represent three types of relationships, namely isEvidenceFor, isAsubclaimOf, and supports. The difference regarding GSN is that the relationship is bottom-up instead of top-down. That is, instead of saying that A is supported by B, it is said that B supports A. Figure 20 shows an example of these elements and relationships to form a CAE argument.

Figure 20. CAE example [19]
On the other hand, as stated before, the OMG’s standardised modelling language SACM 2.0 aims to unify these graphical notations. Table 3 shows the mapping between SACM elements with GSN and CAE, according to the references provided in Annex A: Mappings from Existing Industrial Notations for Assurance Cases by the OMG’s documentation of SACM [16].

Table 3. Mapping between SACM, GSN and CAE

<table>
<thead>
<tr>
<th>GSN Element</th>
<th>CAE Element</th>
<th>SACM Element</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Node Elements:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goal (Rectangle)</td>
<td>Claim (Blue ellipse)</td>
<td>Claim</td>
</tr>
<tr>
<td>Strategy (Parallelogram)</td>
<td>Argument (Green rounded box)</td>
<td>ArgumentReasoning</td>
</tr>
<tr>
<td>Solution (Circle)</td>
<td>Evidence (Magenta rectangle)</td>
<td>InformationElement linked with AssertedEvidence</td>
</tr>
<tr>
<td>Context ( Rounded rectangle)</td>
<td>Side-warrant (Red ellipse)</td>
<td>InformationElement linked with AssertedContext</td>
</tr>
<tr>
<td><strong>Link Elements:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>supportedBy (Filled arrow)</td>
<td>isSubclaimOf (Blue arrow)</td>
<td>AssertedInference from Claim to another Claim</td>
</tr>
<tr>
<td>supports (Green arrow)</td>
<td>AssertedInference from Claim to ArgumentReasoning</td>
<td></td>
</tr>
<tr>
<td>isEvidenceFor (Magenta arrow)</td>
<td>AssertedEvidence</td>
<td></td>
</tr>
<tr>
<td>inContextOf (Empty arrow)</td>
<td>Green arrow</td>
<td>AssertedContext</td>
</tr>
</tbody>
</table>

### 2.3. Argumentation fallacies

As stated in the previous section, the safety argument is an essential part of a safety case and mandatory in many fields for certification purposes. Hence, it is a crucial aspect of those safety-critical systems, where the quality of the presented evidence is as important as the fact that the evidence is actually related to safety requirements. This is where the role of the safety argument acts, so that a correct and quality safety argument is essential for a safe system. According to numerous studies ([14], [20], [21], [30], [31], [36]), faulty reasoning in the safety argument, i.e., a fallacious safety argument, can deteriorate the quality of the system, leading to safety-related failures.

A flawed or fallacious safety argument is the one where the premises that comprise it do not provide enough evidence to guarantee the truth of the conclusion, and thus, is an invalid one.

Damer in [14] argues that there are five criteria of a good argument: Structural, Relevance, Acceptability, Sufficiency, and Rebuttal. The *structural* principle concerns the soundness of its structure, that is, there are no premises that contradict each other or that implicitly assume the truth. The *relevance* principle concerns that the premises must be related and be relevant to the truth of the claim they support. The *acceptability* principle refers to an argument that provides reasons that a mature and rational person should accept with the help of specific guidelines. The guidelines regarding what should be accepted are called “standards of acceptability”, and those regarding what should not be accepted are called “conditions of unacceptability”. For example, the former includes as a standard for accepting a premise “A claim that is a matter of undisputed common knowledge”, and the latter includes as a condition for not accepting a premise “A claim that is self-contradictory or linguistically confusing”, among others. The *sufficiency* principle concerns the amount and weight of the premises that support the conclusion. That is, an argument must have a sufficient number of proper evidence to support the truth of the claim. Finally, the *rebuttal* principle, which states that an argument should include a rebuttal to the possible anticipated criticisms that may be brought against it. Based on this, a fallacy can be defined as the reasoning in an argument that violates any of the above criteria.

Usually, disciplines such as logic and philosophy classify the fallacies either as formal (also called deductive or logical); which can be defined in a standard logic system, or as informal (also called inductive); which do not consider the logical structure of the argument, but the content of the argument. These last ones are the most difficult to detect because, in many cases, the natural language of the arguments allows to dissuade or deceive the reader as well as to camouflage the intentionality of the fallacies. In addition, an argument may not contain any formal fallacy and still not be valid, because although the logic of the structure is valid, the reasoning or the evidence presented in it may be erroneous (informal fallacies). A safety argument can contain both types of fallacies, so a more accurate classification is necessary.
For this, numerous types of fallacies and classifications exist, such as the one proposed by Damer in [14, p. 54], with 12 categories and more than 60 types. However, after an analysis carried out in both general arguments and real safety arguments, Greenwell et al. created a Safety Argument Fallacy Taxonomy [21], composed of common fallacies that can appear in safety arguments. Table 4 shows the complete taxonomy.

Table 4: Taxonomy of Fallacies [21]

<table>
<thead>
<tr>
<th>Circular Reasoning</th>
<th>Anecdotal Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circular Argument</td>
<td>Correlation Implies Causation</td>
</tr>
<tr>
<td>Circular Definition</td>
<td>Damning the Alternatives</td>
</tr>
<tr>
<td>Diversionary Arguments</td>
<td>Destroying the Exception</td>
</tr>
<tr>
<td>Irrelevant Premise</td>
<td>Destroying the Rule</td>
</tr>
<tr>
<td>Verbose Argument</td>
<td>False Dichotomy</td>
</tr>
<tr>
<td>Fallacious Appeals</td>
<td>Omission of Key Evidence</td>
</tr>
<tr>
<td>Appeal to Common Practice</td>
<td>Omission of Key Evidence</td>
</tr>
<tr>
<td>Appeal to Improper/Anonymous Authority</td>
<td>Fallacious Composition</td>
</tr>
<tr>
<td>Appeal to Money</td>
<td>Fallacious Division</td>
</tr>
<tr>
<td>Appeal to Novelty</td>
<td>Ignoring Available Counter-Evidence</td>
</tr>
<tr>
<td>Association Fallacy</td>
<td>Oversimplification</td>
</tr>
<tr>
<td>Genetic Fallacy</td>
<td>Linguistic Fallacies</td>
</tr>
<tr>
<td>Mathematical Fallacies</td>
<td>Ambiguity</td>
</tr>
<tr>
<td>Faith in Probability</td>
<td>Equivocation</td>
</tr>
<tr>
<td>Gambler’s Fallacy</td>
<td>Suppressed Quantification</td>
</tr>
<tr>
<td>Insufficient Sample Size</td>
<td>Vacuous Explanation</td>
</tr>
<tr>
<td>Pseudo-Precision</td>
<td>Vagueness</td>
</tr>
<tr>
<td>Unrepresentative Sample</td>
<td></td>
</tr>
<tr>
<td>Unsupported Assertions</td>
<td></td>
</tr>
<tr>
<td>Arguing from Ignorance</td>
<td></td>
</tr>
<tr>
<td>Unjustified Comparison</td>
<td></td>
</tr>
<tr>
<td>Unjustified Distinction</td>
<td></td>
</tr>
</tbody>
</table>

A safety fallacy is, therefore, a mistake or a flaw in the reasoning of an argument that may go unnoticed or not seem a failure. However, avoiding fallacies during the development phase and detecting them during the revision phase is crucial to avoid future failures in the safety of the system. A description of all categories and fallacies of the taxonomy can be found in [22]. For the purpose of this thesis, only the omission of key evidence fallacy, which is defined below, is focused.

- **Omission of Key Evidence fallacy**

As mentioned in the previous section, there are five criteria to make a good argument, among which is the *sufficiency* one. To meet this criterion, sufficient and not biased evidence need to be provided, as well as the expected evidence, in number and type, for its particular claims.

Therefore, an omission of key evidence fallacy occurs when an argument fails to provide one or more of the expected evidence to justify or support the claim.

2.4. Model-driven engineering

Model-Driven Engineering (MDE) is considered as a promising paradigm in software engineering based on the principle that everything is a model [23]. More specifically, this suggests that to develop a software system, first its model must be developed theoretically, and then transformed it by model transformations into a real entity, that is, an executable code. MDE not only focuses on the model transformations but also the entire process followed to develop software, recognising other important factors such as communication between stakeholders and improving the speed of the response to changes in a system [24]. MDE refers in turn to different development approaches based on modelling software such as Model-Driven Architecture (MDA) and Model-Driven Development (MDD), which are described below. Figure 21 shows a diagram that relates these three model-driven concepts.
To understand this, it is necessary to define what is a model. A *model* is “a representation of a real-world process, device, or concept” [25], that is, an abstraction of specific aspects and relevant details of the structure or behaviour of a real-world system. In order for a model to be processed by tools and correctly interpreted by all stakeholders, it must be formally defined. For that, the structure, terms, notations, syntax, semantics, and integrity rules of the information in the model (i.e., the *modelling language*) must be well defined and consistently represented [26]. A *metamodel*, therefore, defines the abstract syntax of a modelling language.

In the same way, a *meta-metamodel* is defined above it and specifies the underlying layer (i.e., a metamodel). Due to its “infinite” nature and in order to standardise this, the OMG defined the known as “four layered metamodel architecture”, that limit the metamodeling process to four levels; M3 or meta-metamodel level, M2 or metamodel level, M1 or model level, and M0, which represents the system. Although this was retracted in later releases (where the rigidity of this restriction is eliminated, being able to use from a minimum of two layers to those that the user defines), the four-layered architecture was widely accepted and adopted.

In fact, this general metamodel architecture is the one used in the MOF specification and is known as MOF Metadata Architecture, where the upper layer M3 is always MOF. The primary objective of this is that all modelling languages are based on MOF as their meta-metamodel. This is the case for the SPEM 2.0-based process modelling, specified in Figure 1. In addition, Figure 22 shows an example of the general four-layers architecture.

![Figure 21. Relationship between MDE, MDA, and MDE](image1)

**Figure 21. Relationship between MDE, MDA, and MDE [23]**

![Figure 22. Four-layered Metamodel Architecture adapted from [27]](image2)

**Figure 22. Four-layered Metamodel Architecture adapted from [27]**

### 2.4.1. Model-driven architecture

Model-Driven Architecture (MDA) was proposed by OMG in 2001 and encompasses a set of standards with the aim of separating business and application logic from the underlying platform technology, thus allowing technological changes not to affect the results of modelling. For this, “the MDA set of standards includes the representation and exchange of models in a variety of modelling languages, the transformation of models, the production of stakeholder documentation, and the execution of models” [26].

To achieve this separation between the specifications of the system and the platform on which it will be developed, MDA proposes three types of model (namely CIM, PIM, and PSM), which are described below.
• **Computational Independent Model (CIM)** represents the operation of the system as well as the interaction with other systems and with external users. It can also include the requirements, system objectives, laws of a domain, stakeholder needs, etc. The typical way to represent CIM is by Use Case diagrams.

• **Platform Independent Model (PIM)** captures the behaviour of the system, that is, its functionality and structure regardless of the technology and platform to be used. This model is represented by UML and other OMG modelling standards.

• **Platform Specific Model (PSM)** is a model that is defined in terms of a specific platform. PSM complements the PIM model, adding specific technological details of the platform on which it is going to be carried out.

Therefore, the main objective of MDA is to automatically convert a PIM (based on the kernel technologies and focused on the concepts and business requirements) into different PSMs (corresponding with different middleware platforms) [23]. These kernel technologies and middleware platforms can be seen in Figure 23 in the inner part and the middle ring respectively. Also, the outer ring describes the public services provided by MDA, and in the arrows, the high-level domains, that is, the context and purpose of the model.

![Figure 23. Overview of MDA [26]](image)

2.4.2. **Model-driven development**

Another concept related to the model-centric approaches is the Model-Driven Development (MDD). This one is again focused on the model as the primary form of software development, allowing to manage the complexity of it and thereby improve productivity.

The main difference between this and the previous ones is that in MDD the use of OMG standards is not restricted as in MDA, being thus more flexible and allowing to use the one that best suits the user. Besides, it focuses only on the transformation from abstract to concrete, that is, from models to code, without including reverse engineering as, for example, in MDE [24].

2.4.3. **Model transformations**

An essential element for all the concepts described above is the model transformation. According to Kleppe et al. [28], a model transformation is defined as follow:

“A **transformation** is the automatic generation of a target model from a source model, according to a transformation definition. A **transformation definition** is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A **transformation rule** is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language.”

---

Laura Gómez Rodríguez  
A tool-supported method for fallacies detection in process-based argumentation

---

23
Therefore, a model transformation can be defined as a set of rules that define how to automatically map a model in a source language into another model in a target language. This allows converting one type of model into another, a model into concrete implementation of code, or defining a model from specific source code, through the following types of transformations:

- **Model-to-model (M2M)** transformation converts a source model (e.g., a PIM) into a target model (for example another PIM or a PSM). Both source and target models can have the same or different metamodel. To this end, there are numerous model-transformation languages, including ATLAS Transformation Language (ATL), Query/View/Transformations (QVT), Janus Transformation Language (JTL), or Epsilon Transformation Language (ETL), which will be used in this thesis.

- **Model-to-text (M2T)** transformation, also known as model-to-code (M2C) or code generation, converts an abstract model into a specific entity of code. Some of the technologies and frameworks that provide M2T transformation are Acceleo, Xtend, Xpand, JET or MOFScript, among others.

- **Text-to-model (T2M)** transformations allow reverse engineering, that is, to extract a high-level model from a specific code.
3. Related work

In order to make the process of assurance and review of safety arguments more trustworthy, and mainly, less costly and time-consuming, many approaches have been proposed to try to detect and avoid fallacies in such safety arguments. Kelly presents in [29] a systematic review process based on four steps, which are: (1) Argument comprehension, (2) Well-formedness checks, (3) Expressive sufficiency checks; and (4) Argument criticism and defeat; where the elements of rebuttal and undercutting are included. In [30], Yuang and Kelly propose an argument schemes approach to: (i) avoid fallacies by constructing arguments through a series of ten argument schemas and templates; and (ii) detect fallacies once the argument is developed through a set of critical questions (CQs), which reveal weaknesses or flaws in the argument if they cannot be adequately answered. In another paper [31], Yuang et al. propose a Safety Argument Review Model (SARM) and a dialogue-based tool for safety arguments review (DiaSAR). This tool, through a graphic interface, allows the “argument proposer” (i.e., who creates and defends the argument) to connect with the “reviewer” (i.e., who questions and rebuts the argument, as well as proposes counter-arguments). However, the quality of the reviewed arguments depends on the knowledge and expertise of the reviewers, and therefore its correctness cannot be guaranteed.

Although these kinds of tools and techniques make the process easier, both for system engineers to develop the argument and for the other stakeholders to review and evaluate it, they are informal logic argument schemes. In a closer step to automation, in [32], [33], [34], and [35], different approaches to formalising the safety arguments are proposed. In [32], Groza and Marc propose an ontology that formalises GSN, translating thereby the arguments represented in GSN into description logic, specifically in $ALC$ (Attribute Concept Language with Complements). In addition, this approach is implemented as a tool for possible automation. On the other hand, Rushby proposes in [33] the representation of arguments in a classical formal notation, in particular, in the higher-order logic one (HOL). He states that the elements of the argument can be described in any modelling formalism able to use “Satisfiability Modulo Theories (STM) solvers” (e.g., PSV), supporting thus strong and automated deduction methods such as theorem proving and model checking. In another paper [34], Rushby proposes an improvement of this approach by adding “defeater” predicates to the premises of the formalised cases. These “defeaters” –also known as “undercutting”, “undermining” and “rebutting”– are initially set to false, and the reviewer can activate it to see the consequences, as well as add additional premises and restrictions. Finally, Brunel and Cazin [35] proposed: (a) a framework to represent an argument in ForSALE language (Formal Safety Argumentation Language and Environment), and (b) a formal semantics that allows automatic validation of the argumentation, formalising claims and sub-claims in Linear Temporal Logic (LTL). These approaches try to automate the verification of the argument by using formal logic.

Similarly, a predicate logic-based approach is presented in [36]. This approach differs from the previous ones since it is based on the formal representation of the sentences contained inside the GSN nodes, not the entire argument. For that, they propose an ontology that contains a set of constant, function, and predicate symbols, which create the expressions of GSN nodes. The vocabulary ontology was derived from a domain analysis of existing safety cases, based on the frequency of relevant keywords that appears in them, so the search function only works if the words used in the argument are stored in the database. The fallacies consider in this approach do not include the omission of key evidence.

Despite this, Sokolsky et al. claim in [37] that “complete formalization of the claims in an assurance case is unlikely to be achieved”. Graydon also claims in [38] that a complete formalisation would limit the reading audience to those who can read formal logic and would make the argument less comprehensible and read-friendly. Besides, after a systematic survey and analysis of the literature, he states that there is no evidence of such formalisation helps to avoid or detect the informal fallacies, for which a human review is needed.

Greenwell and Knight [20] present a systematic approach called Pandora for analysing safety-related digital system failures based on the concept of safety cases. They state that such digital failures usually indicate the presence of fallacies in the related safety arguments so, the approach derives evidence from a failure to discover safety fallacies, and then propose recommendations for addressing the discovered fallacies, generating a revised safety argument. However, this approach begins with the pre-failure safety case, so it depends on the completeness of a safety case and cannot be used to prevent the fallacies.

Ayoub et al. present in [39] a systematic approach to construct confidence arguments, that is, those whose overall confidence is considered acceptable, and to identify and manage the weaknesses associated with the software safety arguments. They propose a common characteristics map, that is, a structured collection of the common concerns about the trustworthiness (e.g., tool qualification for tool-derived evidence), to provide guidelines for generating positive confidence arguments. In another paper [40], they present a structured approach for assessing the overall level of sufficiency of safety arguments. For that, they use the concepts of belief
combination and basic probability assignment to calculate the overall sufficiency—or insufficiency—of the safety arguments, based on a measure of sufficiency—and insufficiency—for each node.

Denney and Pai [41] have developed a toolset for assurance case automation, named AdvoCATE. This tool supports the automatic creation of assurance arguments, the integration of formal methods, and an automatic pattern instantiation (either interactively or by the data extracted from tools of hazards and safety requirements analysis), among others. In addition, it supports verification of the arguments, specifically structural properties (e.g., no cyclic links, internal completeness), that is, it checks the soundness and well-formedness of the argument, but not its contents.

However, most of these papers either focus on the verification of the overall structure of the argument (structural principle), or are based on the expert opinion and manual generation and review. In contrast, our approach provides the detection of the omission of key evidence fallacy (sufficiency principle) in process models, and the automatic generation of fallacy-free process-based safety arguments from these models, without the need for a complete safety case, allowing thus the prevention of committing such fallacies.
4. Problem formulation and analysis

4.1. Problem formulation

As described in Chapter 1, the generation of safety cases is mandatory in many standards and essential for certification purposes in the development of safety-critical systems. In particular, the creation of a well-defined, clear, and quality safety argument, which aims to demonstrate that a system is safe to operate in a particular context. Currently, the creation of safety argumentation is a very time-consuming and costly process. Because this task depends on the experience and knowledge of the developer, it is also error-prone. Inappropriate reasoning in the argumentation can undermine the credibility of the argument and lead to safety-related failures in the system. For all of this, the avoidance, detection, and removal of fallacies in the argumentation is an essential task to reduce the risk of failures in a safety-critical system.

This thesis aims to develop a method for fallacies detection in process-based argumentation. More specifically, this thesis is defined in the scope of the ongoing research project AMASS. To reach this aim, an analysis of this problem is performed in the next section by formulating a series of research questions, which are explained below.

4.2. Problem analysis

The main goal of this thesis is the achievement of a tool-supported method that incorporates assistance in the process-based arguments generation for avoiding and detecting safety fallacies. To accomplish this goal, we have formulated the following research questions:

RQ1: How do we detect the omission of key evidence fallacy in process-based arguments?
This is the main problem which we are trying to address through this thesis. Section 4.2.1 analyses this question and divides the problem into subproblems.

RQ2: Can we provide process compliance with argumentation?
This question presents the second problem that we are trying to address with this work. A more in-depth analysis of this question is carried out in Section 4.2.2.

RQ3: Can we increase the quality of safety arguments?
Once both previous research questions have been solved and the developed solution is evaluated, we will be able to determine whether through our solution, the quality of safety arguments have been improved. This will be discussed in Chapter 7 Section 5 along with the results of the evaluation of the solution.

4.2.1. Detect the omission of key evidence fallacy problem

The detection, prevention and avoidance of fallacies should be done in the first steps of the safety case development in order to be more efficient and avoid extra costs both in time and money. In this way, if there is some evidence that is omitted and thus, the risk of creating a weak safety argumentation, this can be fixed before starting any process of transformation or creation of the argument. To achieve the development of the fallacy detection algorithm the following questions have been addressed:

- **Which modelling tool is used?**
  The first step is to select the tool in which the safety process is going to be modelled.

- **Which process elements are used to identify the omission of key evidence fallacy?**
  The modelled safety process is composed of many types of elements that need to be certified in order to provide the sufficient evidence to support the claim. Due to time constraints, as well as the scope of the project in which this thesis is defined, the implementation of all these elements is either not possible or not available, and therefore, a selection of some of them is needed.

- **How we model the standard requirements and the process elements to detect the omission of key evidence fallacy?**
  To verify that the required evidence is provided we have to figure out a solution of how to model and map standards requirements and process elements. Moreover, after selecting the process elements that are going to be analysed to find a lack of sufficiency in the form of provided evidence, it is also necessary to establish how they are modelled. That is, which of their properties must be completed and with what.
• How we implement the detection of the omission of key evidence fallacy?
  Finally, a match algorithm between the standard requirements and the process evidence is necessary to determine either if the provided evidence is sufficient or an omission of key evidence fallacy is detected.

4.2.2. Transform the validated process into a safety argumentation problem

As stated in previous sections, the creation of a process-based argumentation is a costly, time consuming and error-prone task, but also essential to ensure the quality, safety confidence, and compliance of the system. Therefore, an automatic transformation from safety process into process-based argumentation is essential to facilitate and ensure that task. This process was already implemented in a previous transformation plugin. This one takes an XML file (obtained through the export functionality of EPF Composer) as source model to generate the transformation from OpenCert tool [43]. However, in order to make this process more efficient, there is a need to provide the activation from EPF Composer. This functionality was not possible to implement in the previous transformation plugin since EPF Composer had not been migrated to newer Eclipse versions. This issue has been solved recently (the migration from Eclipse Galileo 3.5.2 to Eclipse Neon 4.6.3) as Javed and Gallina state in [44], so the activation of this transformation process from EPF Composer can be now addressed. To reach this subproblem, the following questions are stated:

• Which tools are involved in the process in the context of AMASS project?
  Identify the tools and analyse the metamodels in which they are based will allow us to understand how the source and target models are structured. This is essential to develop the M2M transformation.

• How we map the source model elements to the target model elements?
  A mapping between the different entities of the involved metamodels has to be defined before writing or modify the transformation algorithm.

• Which model transformation language is used?
  Based on the specific needs of the transformation, a proper transformation language must be selected for providing the rules and other transformation scenarios or processes and mappings needed.
5. Method

This chapter presents the methods that have been used for both the overall approach of the thesis’ work and to resolve the specific subproblems stated in Chapter 4, Section 2. Section 5.1 explains the research methodology selected to follow the workflow of the thesis, and Section 5.2 explains the specific methods used to solve the problem.

5.1. Research methodology

The research methodology could be defined as a systematic method or approach that comprises defining a problem, finding a solution, collecting and evaluating data, and extracting conclusions.

As previously stated, the main objective of this thesis is detecting and avoiding safety fallacies in the generation of process-based argumentations (through a tool-supported method). For reaching this, a proper research methodology was defined, based on the qualitative nature of the objectives. A case study with real data extracted from the ECSS-E-ST-40C standard was also performed to validate and evaluate the applicability of the proposed solution. Figure 24 shows the research methodology that we followed taken from [45].

![Research methodology adapted from [45]](image)

The first part of the methodology (I) corresponds to define and understand the research problem (e.g., what are the safety fallacies, which ones could be automated? etc.), including the understanding of the project context (such as safety cases, AMASS project). After that, a review of the related literature, both needed background concepts, and previous research findings and related work, is performed (II). This review of the literature is carried out mainly through research papers, but also with books, web pages, and official documentation, all of them properly documented in the References section. Once the problem is defined, and the literature studied, the specific research questions are formulated (III). Through these, we expect to contribute to the knowledge of the thesis domain by answering the research problem. In addition, the main problem is divided into smaller sub-problems in order to manage it in more specific and reachable goals. Then, the design of the research (IV), including the design of the case study and the methods to solve the problem is performed. This also serves as criteria for the evaluation of the solution, to figure out whether the proposal achieves the sub-goals. After that, the solution to the research problem is designed and implemented (V), as well as evaluated and validated (VI) to ensure that it meets with the design requirements established in step IV. Finally, the obtained results are interpreted, and the overall work is documented in this thesis report (VII).

5.2. Methods to solve the problem

In order to achieve the main goal of this thesis (corresponding to the solution in the above flowchart), and following the sub-goals presented in Chapter 4, two activity diagrams have been developed, which represent the methods used to achieve the corresponding sub-goals.

These methods are found after answering the questions presented in Chapter 4, Section 2. Specifically, Section 5.2.1 answers the questions related to the fallacy detection sub-goal, and presents the proposed method to
solve it. Section 5.2.2 answers the questions regarding the transformation problem and explains the proposed method to address it.

5.2.1. **Detect the omission of key evidence fallacy method**

This section answers the questions raised in Chapter 4. In addition, an activity diagram of the method used to solve the problem of the fallacy detection is presented in Figure 31, and the deepest explanation of each of the followed activities is provided below.

- **Which modelling tool is used?**

Since this thesis is defined in the scope of the AMASS project, the selected modelling tool is the EPF (Eclipse Process Framework) Composer [46], which is implemented within the AMASS platform [47]. EPF Composer is an open source tool that “aims at producing a customizable software process engineering framework, with exemplary process content and tools, supporting a broad variety of project types and development styles” [46]. These qualities make the EPF Composer suitable for our purpose.

This tool is based on the Unified Method Architecture (UMA) metamodel, which is an unification and evolution of different method and process engineering languages, including SPEM 2.0 (presented in Chapter 2). Therefore, the source metamodel in the transformation is UMA. This metamodel describes, in a clearly separated way, how the Method Contents elements and its application in Processes are defined. Figures 25 and 26 shows the organisation diagram of the Method Contents and Processes respectively.

![Figure 25. UMA metamodel of the Method Content [46]](image)

The transformation process is invoked from a Delivery Process element, and involves the following process elements: Delivery Process, Capability Pattern, Phase, Activity, Task Descriptor, Role Descriptor, and Work Product Descriptor. These elements have been described in Chapter 2, Section 1 under the SPEM 2.0 process elements, and their mapping to the target model elements are specified in the next question. Other process elements, such as Breakdown Element and Process Element, are also used in the transformation to perform certain operations, which are explained in Chapter 6, Section 2.1. The inheritance relationship between these process elements is defined in Figure 26.
• Which process elements are used to identify the omission of key evidence fallacy?

Since within AMASS project, this thesis is focused on the planning phase of the processes, some elements such as Work Products are not available in those stages of the projects. Process-based argumentation at planning phase aims to assure that each phase or activity has been planned, and after the approval of the plan, the real development of such Work Products (as outcomes of activities/tasks) begins. Therefore, evidence related to this kind of elements cannot be presented, so we will not take them into account.

In addition, although all the remaining process elements must be validated to ensure the no omission of key evidence fallacy in the construction of the argumentation, due to time limitations only two of them have been selected. The first one is the Role. As stated in Chapter 2, a Role in a safety process has a series of responsibilities to perform critical and relevant tasks, and therefore, must possess certain key competencies. Not to provide certifications for these key competencies results in an omission of key evidence fallacy. The second one is the Tool Mentor. These elements are related to Tool elements, in the form that a Tool Mentor describes how to use a specific Tool to perform a certain activity. These Tools, and therefore, its related Tool Mentors must be qualified to ensure that the evidence generated by that tool is trustworthy and valid. As in the case of the roles, the no provision of certifications for these required qualifications results in an omission of key evidence fallacy.

• How we model the standard requirements and the process elements to detect the omission of key evidence fallacy?

The selected technique to model the standard requirements is obtained from [48], an IBM article that describes the best practices for mapping a process to a standard with IBM Rational Method Composer (RMC) [49], the commercial version of EPF Composer, and from [59]. The process consists of the following steps:

(i) Capturing standard requirements: In order to map a process to a standard, we have to represent the standard’s requirements as elements. To do so, in RMC they create a user-defined type called Standard Requirement, where capture the standard requirement in its Brief description field. Nevertheless, EPF Composer does not support the user-defined type. Therefore, to do so, a guidance type Practice called requirement is customised with an icon in a separate plugin (compliance_modeling), as described in [59]. Figure 27 shows an example of this.

In addition, to capture the standard requirements in itself, another new plugin is created (standard_requirements). Within this, a Practice element is created to capture each requirement in its Brief description field, and using the Variability type “Extends” to extend the previously created requirement element, as shown in Figure 28.
Modelling process life-cycle: In a separate plugin (process_lifecycle), modelling the elements of the process in the Method Content package (e.g., roles, tasks, work products) and developing the life-cycle of the safety process under the Processes package (e.g., phases, activities, task descriptors) as shown in Figure 29.

Mapping standard requirements: Finally, in a mapping_requirements plugin, mapping the standard’s requirements with the process elements through the Variability type “Contributes” and the References tab. To do so, for each captured requirement we have to create a new Practice that contributes to the corresponding original requirement in the standard_requirements plugin, and links it with the corresponding process element in the process_lifecycle plugin through the Content elements field in the References tab. Figure 30 shows an example of this.
In addition to this, the properties used to model the process elements, according to the stated in the question “Which process elements are used to identify the omission of key evidence fallacy?”, are the following:

- **Skills (Staffing Information section)** for the **Roles**. Certifications against the required standard’s requirements are presented in this field. If a **Role** has more than one requirement to fulfil, the presented pieces of evidence or rationales that correspond to different requirements shall be separating by a semicolon (e.g., “Certification against: requirement 1; requirement 2; requirement 3”).

- **Key considerations (Detail Information section)** for **Tools**. Certifications or rationales against required tool qualifications shall be defined in this field. Besides, the associated **Tool Mentors** have to be linked to the **Tool** through the **Tool Mentors** tab.

- **How we implement the detection of the omission of key evidence fallacy?**

Based on all previously established, the **omission of key evidence** fallacy in process-based arguments is detected through the comparison of the modelled standard’s requirements and the process elements related to them. This functionality is implemented in an Eclipse plugin, and the details of the matching algorithm are explained in the next chapter.

Figure 31. Activity diagram for fallacy detection method

1. **Study of SPEM 2.0 and EPF Composer**. The study of the SPEM 2.0 through its OMG specification and related literature, as well as the study and training of the EPF Composer through its User Manual and some tutorials to get confidence in how the process elements are related and how the safety process are properly modelled.

2. **Analysis of standards mapping techniques**. Looking for the best practices for mapping a process to a standard in order to make the implementation of the algorithm suitable to the detection of the **omission of key evidence** fallacy.

3. **Capturing standard requirements and modelling the process**. An analysis of the standard document to determine which specifications should be modelled as requirements in the process, and the implementation of the model in EPF Composer.

4. **Design the fallacy detection algorithm**. To determine if there are an **omission of key evidence** fallacy, the algorithm checks if the provided pieces of evidence (in the form of skill certifications and tool qualifications) are enough to satisfy all the captured requirements of the standard.
5. **Implement the algorithm in Java.** Implementing the previous design in Java programming language within an Eclipse plugin. To check the correctness of the algorithm, a safety process with multiple scenarios is tested with the fallacy detection algorithm. If the obtained outcome is not satisfactory, the process is repeated, skipping the unnecessary steps, in order to improve the results.

5.2.2. **Transform the validated process into a safety argumentation method**

This section answers the questions raised in Chapter 4 regarding the transformation problem. In addition, an activity diagram of the method used to solve such problem is presented in Figure 33, and the further explanation of the activities is provided below.

- **Which tools are involved in the process in the context of AMASS project?**

As stated above, the selected tool used to model the safety process (i.e., the source model) is EPF Composer. This tool, as well as its metamodel, are explained in the previous section.

The second tool involved in the process is OpenCert [50], since the model generated by the transformation (i.e., the target model) is represented –using the GSN notation– through the OpenCert assurance case editor. These models are compliant with the Common Assurance and Certification Metamodel (CACM), which includes the OMG’s standard SACM metamodel, introduced in Chapter 2, Section 2. Some alterations to the SACM metamodel such as including concepts for Modular Argumentation (Module / Patterns, Contract / Agreements) have been made [50].

The main elements of the Argumentation Metamodel have been defined in Chapter 2, Section 2.2 under the GSN elements and their mapping in Table 3, and an overview of the metamodel structure is shown in Figure 32.

- **How we map the source model elements to the target model elements?**

A mapping between the elements of the source (process elements) and target (argumentation elements) models involved in the transformation is described in Table 5. The mapping has been performed following the Work Breakdown Structure of the EPF Composer processes. Specifically, the Delivery Process, that contains all the information about the process, is transformed into a Case. Then, the Capability Pattern is mapped into the top-level Claim and is composed in turn of sub Claims corresponding to the Phases, Activities, and Tasks in which the process is structured. Also, for each set of elements related to the Task (Roles, Work Products, Guidances), an ArgumentReasoning is provided, as well as a Claim for the requirements of those elements. Finally, the presented evidence against the standard’s requirements and the rationales not to present them, are mapped into InformationElementCitation, Property type=“solution” and Property type=“justification”, respectively. The mapping follows the MDSafeCer principles proposed by Gallina in [11].

<table>
<thead>
<tr>
<th>Table 5. Mapping elements</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SPEM/UMA</strong></td>
<td><strong>SACM/CACM</strong></td>
</tr>
<tr>
<td>DeliveryProcess</td>
<td>Case</td>
</tr>
<tr>
<td>Capability Pattern, Phase, Activity, Task Use (Task Descriptor), Role Use (Role Descriptor), Work Product Use (Work Product Descriptor), Guideline, Checklist, Example, Tool Mentor</td>
<td>Claim</td>
</tr>
<tr>
<td>A set of Phases, Activities, TaskUse, RoleUse, WorkProductUse, Guidelines, Checklists, Examples, Tool Mentors</td>
<td>ArgumentReasoning</td>
</tr>
<tr>
<td>Evidence associated with RoleUse (skills certification), Tool Mentors (tool qualification), WorkProductUse, Guideline, Checklist, Example</td>
<td>InformationElementCitation, Property type=“solution”</td>
</tr>
<tr>
<td>Rationales associated with the omission of RoleUse evidence (skills certification), and Tool Mentor evidence (tool qualification)</td>
<td>InformationElementCitation, Property type=“justification”</td>
</tr>
<tr>
<td>Delivery Process’s Purpose field</td>
<td>InformationElementCitation, Property type=“context”</td>
</tr>
<tr>
<td>Relationship between Phases, Activities, TaskUse</td>
<td>AssertedInference</td>
</tr>
<tr>
<td>Relationship between a RoleUse or a Tool Mentor and their evidence</td>
<td>AssertedEvidence</td>
</tr>
</tbody>
</table>
Which model transformation language is used?

Since the transformation process is an update of an existing one, the selection of the language was limited to the language that the existing one previously used, which is the Epsilon Transformation Language (ETL) [51]. ETL is a hybrid language that provides both a declarative rule-based transformation language and imperative characteristics to address complex and interactive transformations.

In addition, ETL allows transforming "an arbitrary number of source models into an arbitrary number of target models" [52], not only a single source model into a single target model. This is especially important since the Eclipse Modeling Framework (EMF) persistence implementation of the EPF Composer stores the container Method Library and all its contents elements in a repository of separated models (XMI files). Therefore, the transformation from the EPF process into a safety argumentation involves several source models, and ETL enables this.
1. **Study of GSN and SACM/CACM.** A study of the graphics-based notation for safety case representation GSN through related literature, and the OMG’s standard SACM metamodel (CACM in OpenCert) is carried out before starting the M2M transformation.

2. **Understanding existing mapping and implementation.** The understanding of the existing mapping between EPF processes elements (SPEM/UMA elements) and GSN and SACM/CACM concepts, as well as their properties (such as id, name, description). In addition, the study of the existing implementation is carried out to be able to update it. This implementation is compliant with MDSafeCer [11] principles, and was implemented in the first prototype of AMASS platform (see Training on the Prototype P1: WP6 Session in [47]).

3. **Updating the existing implementation.** Updating the transformation with the new concepts such as the evidence and justifications of the roles and tool mentors, among others. Besides, the update of the argument generator plugin implementation is developed to transform automatically, from the EPF Composer, the process modelled into both safety argumentation model and diagram and persist them into the CDO Repository.

The iteration of the needed activities is repeated until the model and diagram obtained through the transformation are correct.
6. Solution

This chapter presents the solution developed for the problem stated in Chapter 4. The solution has been reached through the methods defined in Chapter 5, and its applicability is analysed through a case study detailed in Chapter 7. Section 6.1 presents the solution obtained for the problem of fallacy detection, and Section 6.2 presents the solution for the transformation problem.

6.1. Detect the omission of key evidence fallacy solution

This section details the solution obtained to detect and prevent the omission of key evidence fallacy in process-based arguments. To do this, the detection of such fallacy is carried out in the first step of the process, i.e., before transforming and creating the argumentation. The validation is performed on the safety process, specifically on either the Capability Pattern or the Delivery Process, and checks that all the required pieces of evidence not to commit an omission of key evidence fallacy in the argumentation are presented. That is, the validation process compares every presented evidence against the standard's requirements related to them. After that, the detailed validation results and (in case critical information is omitted) the recommendations to improve the safety process are both printed on the console and stored in a validation report in the form of TXT file. In this way, a lack of key evidence for supporting the top-level claim of the process can be solved—through remodelling the process—by the process engineer on the earlier stage of the safety argument creation, without involving the transformation process. Figure 34 shows the main workflow of the process, and Section 6.1.1 explains and details the most important parts of its implementation in Java programming language. Section 6.1.2 presents the plugin that implements the Fallacy Detection functionality.

![Figure 34. Flowchart of fallacy detection process](image)
6.1.1. Java implementation

In order to achieve the tool-supported method to detect fallacies, the previous concepts have been implemented in Java programming language. A general description of the implementation, as well as the main functionalities and methods, are described below.

As stated before, the validation process compares the presented evidence against the standard's requirements related to it. To do so, we can divide the fallacy detection algorithm into three parts or functionalities: the extraction of the requirements, the extraction of the process elements, and the matching or comparison between them.

**Extraction of the standard's requirements implementation**

This functionality consists in extracting all the modelled requirements from the standard_requirements plugin and storing them properly. To do so, starting at the current Method Library, all the modelled Method Plugins are obtained. After that, we look for the Practices—and sub-practices—(through going down the internal structure of the process) in order to find the standard’s requirements (modelled as Practices as specified in Section 5.2.1). Once the Practices elements are reached, we have to identify if they belong either to the staffing plan requirements or the tool qualification plan requirements. If the sub-practice belongs to the staffing plan, its referenced element is going to be a Role type element. On the contrary, if the sub-practice belongs to the tool qualification plan, its referenced element will be a Tool type element. Therefore, the last step is to get its referenced elements and check what type of element they are. Depending on the obtained type, the Practice is stored either in a “Roles list” or in a “Tools list”, so they can be used by other functionalities of the fallacy detection algorithm.

The implemented method to obtain these Practices is presented in Figure 35. For this purpose, we start with the obtention of the current Method Library through the LibraryService utility. Once this is obtained, we get the next contents, that is, the Method Plugins, the System Packages, the Content Packages (belonging to the Method Contents), and finally the Practices (through the Content Elements of the Content Packages). For the Method Plugins, we obtain all of the Method Library’s ones (line 4), and then we go through them until reaching the one with the standard’s requirements. In order to not repeat this process unnecessarily, a “flag” is checked so that once the plugin containing the standard’s requirements has been found, the iterations over the remaining Method Plugins finish. For the System Packages, we know that the Method Content Elements are in the “CoreContent” one, so we limit the System Package to that one (line 14). After that, we obtain the Content Packages and the main elements, specifically the Practices and their sub-practices.

```java
MethodLibrary src = LibraryService.getInstance().getCurrentMethodLibrary();
Boolean flag = false;
//Get the Method Plugins
List<MethodPlugin> listPlugins = src.getMethodPlugins();
for(MethodPlugin plugin : listPlugins){
    if(flag){break;}
    //Get the System Packages
    listSystemPackages = TngUtil.getAllSystemPackages(plugin);
    for(Object sp : systemPackages){
        if(flag){break;}
        if(sp instanceof ContentPackage){
            //Get the Content Packages
            ContentPackage systemPackage = (ContentPackage) sp;
            if (systemPackage.getName().equals("CoreContent")){
                List<MethodPackage> methodPackages = systemPackage.getChildrenPackages();
                for(MethodPackage methodPackage : methodPackages){
                    if(flag){break;}
                    ContentPackage contentPackage = (ContentPackage)methodPackage;
                    List<ContentElement> contentElements = contentPackage.getContentElements();
                    for(ContentElement contentElement : contentElements){
                        //Get the Practice
                        if(contentElement instanceof Practice){
                            Practice practice = (Practice) contentElement;
                            //Get the Sub Practices
                            List<Practice> subPractices = practice.getSubPractices();
                        }
                    }
                }
            }
        }
    }
}
```

Figure 35. Extraction of standard's requirements implementation

**Extraction of the process elements implementation**

Once the standard’s requirements have been obtained and stored, we extract the modelled process elements from the process_lifecycle plugin. To do so, the first step is to obtain the process element selected by the user (Capability Pattern or Delivery Process), implemented through the selection service provided by the Eclipse
A tool-supported method for fallacies detection in process-based argumentation

workbench as shown Figure 36. Then, with an Iterator, we go through their different eContents (MethodElementProperty, etc.) until reaching the ProcessPackage, that is, the selected process itself.

```java
1  ISelection sel = page.getSelection();
2  TreeSelection tree = (TreeSelection) sel;
3  ProcessComponentImpl process = (ProcessComponentImpl) tree.getFirstElement();
4  for (Iterator it = process.eContents().iterator(); it.hasNext(); ) {
5      Object nextObj = it.next();
6      if (nextObj instanceof ProcessPackage)
7          (ProcessPackage) nextObj : null;
8          if (child == null) {
9              continue;
10          }
11          //Get the selected element (Delivery Process / Capability Pattern)
12          if (child instanceof ProcessPackage){
13              ProcessPackage proc = (ProcessPackage) child;
14          }
```

Figure 36. Get selected process element implementation

Once we have the selected process, we use its internal structure to reach the method elements involved in the detection of the omission of key evidence fallacy (i.e., Roles and Tools in this approach). Figure 37 shows the implementation of this method in Java programming language. For this, the Work Breakdown structure of the process is followed, obtaining first all the process elements from the selected Process (line 2). Then, the private method getBreakdownElements() is used to go through the different process elements (Capability Patterns, Phases, Activities, etc.) until the Task Descriptors are reached.

```java
1  //Get the Process Elements (Capability Patterns for Delivery Process / Capability Pattern)
2  list(ProcessElement) listProcessElements = proc.eProcessElements();
3  //If the selected element is a Delivery Process, for each Capability Pattern
4  if (proc instanceof CapabilityPattern){
5      //Get the Phases
6      list<BreakdownElement> listPhases = ((CapabilityPattern) proc).getBreakdownElements();
7      for (BreakdownElement phase : listPhases){
8          //Get the Activities
9          list<BreakdownElement> listActivities = ((Phase) phase).getBreakdownElements();
10         for (BreakdownElement bde : listActivities){
11             if (bde instanceof Activity){
12                 //Get the tasks
13                 list<BreakdownElement> listTasks = ((Activity) bde).getBreakdownElements();
14                 for (BreakdownElement bde2 : listTasks){
15                     if (bde2 instanceof TaskDescriptor){
16                         //Get the Roles that perform that Task
17                         TaskDescriptor taskDescriptor = (TaskDescriptor) bde2;
18                         list<RoleDescriptor> listRoles = taskDescriptor.getPerformersBy();
19                         listToolMentorsTask = listToolMentorsTask.add(taskDescriptor.getPerformersBy());
20                         listToolMentorsTool = listToolMentorsTool.add(taskDescriptor.getPerformerBy());
21                    }
```

Figure 37. Get process elements implementation

In addition, the information needed for the validation, that is, the Skills for the Roles, the Key Considerations for the Tools, and the Brief description for the requirements, is obtained. To do so, for each Task Descriptor of the process, we obtain the involved Roles (through its defined methods getPerformedBy(), getAdditionallyPerformedBy(), and getAssistedBy()), and the associated ToolMentors (getToolMentors()) in order to validate them.

To obtain the Skills of the Roles, a new Iterator is implemented to derive the RoleDescription class of the Role (where the Skills field is located), and once we have the evidence for the Role, we look for its related Practice with the requirements.

For the ToolMentors case, since the qualifications are related to the Tools instead of the Tool Mentors, to get the evidence we have to obtain its related Tool. Figure 38 shows the implementation of this. To do so, first we extract the Tool Mentors involved in the task and add them in a list (line 3). Then, we obtain the Tools associated with the process from the reference elements of the practices under the name “Tool qualification plan” (obtained in the first functionality). After that, from these Tools we obtain their related Tool Mentors and add them in a new list (line 9). Then, we compare both lists (listToolMentorsTask and listToolMentorsTool) to check if the
ToolMentor of the Task has any associated requirement. If it has, we use an Iterator again to obtain the ContentDescription class and from there, its qualification (line 23).

Finally, once all the needed information has been extracted, the designed method to check and validate the omission of key evidence fallacy (detailed below) is invoked.

```java
1  //Get the Tool Mentors of the task
2  Task task = taskDescriptor.getTask();
3  List<ToolMentor> listToolMentorsTask = task.getToolMentors();
4  if(!listToolMentorsTask.isEmpty()){  
5      //for each Tool of the requirements, get the Tool Mentors
6      for (ContentElement refTool: listReferencesTools){
7          String qual="";
8          Tool tool = (Tool) refTool;
9          List<ToolMentor> listToolMentorsTool = tool.getToolMentors();
10         //and check if the associated Tool of the Tool Mentor of the task has any requirements
11         for(ToolMentor tm: listToolMentorsTool){
12             //if it has
13             if(tm.getUid().equals(tm2.getUid())){
14                 for (Iterator iter = toolContents.iterator(); iter.hasNext();){
15                     Object nextObject = iter.next();
16                     ContentDescription toolDescription = nextObject instanceof ContentDescription ?
17                         (ContentDescription) nextObject : null;
18                     if (toolDescription == null) {
19                         continue;
20                     }
21                 }
22                 //get the qualification
23                 qual = toolDescription.getKeyConsiderations();
24             }
25         }
26         //get the qualification
27         qual = toolDescription.getKeyConsiderations();
28     }
```

**Figure 38. Get tool qualifications implementation**

**Matching algorithm implementation**

Finally, a private method is implemented to compare the extracted information, that is, to check whether the omission of key evidence fallacy occurs. For this, it should be noted that just with the omission of a single piece of evidence, an omission of key evidence fallacy is considered. In addition, the possibility of incorporate rationales to “omit” justifiably a required evidence must be taken into account. Once these clarifications are defined, the matching algorithm works as follow:

The method receives the requirements and evidence extracted (through the previous parts of the implementation) from the element under validation, and its ModelObject type (i.e. “Role” or “Tool”), considering in this way, a distinction between the type of the elements. This is carried out through a switch statement, switching depending on the type. With this, the procedure to validate the Role type elements is separated from the one to validate the Tools, and thereby, if more elements to validate the omission of key evidence fallacy are implemented in future work, they can be easily added in additional “cases”.

Firstly, we check that there is at least one evidence –or rationale–, in particular, that the evidence field is not empty and we can check the match with the obtained requirements. Otherwise, we directly mark as an omission of key evidence fallacy and move on to the next element. In case that the presented evidence is not empty, we split the obtained requirements (since as stated in Section 5.2.1, every single requirement relating to the same element is separated from each other by a semicolon). Then, we ensure that the obtained content is a requirement, specifically, that it does not have any additional information such as an introductory sentence. Figure 39 shows the implemented method to remove this undesired information. To do so, we go through the requirements (character by character, with the assistance of a StringBuffer) until finding either a colon or a key construction that determines an introduction sentence. From this character, the remaining part of the content is considered as the requirement and replaces the previous one.

```java
1  arrayRequirements = requirements.split("; ");
2  //Remove the introduction sentence, e.g. “Should be competent and have relevant experience: Shall be
3  StringBuffer stringBuff = new StringBuffer();
4  for (int i=0; i<arrayRequirements[i].length(); i++){
5      char c = arrayRequirements[i].charAt(i);
6      String substr = stringBuff.append(c).toString();
7      if(substr.contains(":"))||substr.contains("competent in")||substr.contains("shall be able to")|
8         String req = arrayRequirements[i].substring(i+1, arrayRequirements[i].length());
9         arrayRequirements[i] = req;
10        stringBuff.delete(0, stringBuff.length());
11        i=0;
```

**Figure 39. Remove additional information implementation**
Removing and cleaning the prepositions

When the requirements are obtained and adapted, the split by semicolons is done again for the evidence. Then, once we have the “requirements list” and the “evidence list” (which contain in each position an individual requirement and a piece of evidence, respectively), we remove the prepositions, articles, and other non-main words of them in order to obtain a greater accuracy when comparing the requirements with the evidence. That is, since this approach is based on the number of words that match between the evidence and the requirement to determine if the presented evidence is correct, the presence of articles and prepositions could derive in a greater (and non-real) percentage of match. For example, if we do not remove the non-main words, and we have the following situation:

Requirement: “Experience with Linux and Python programming language.”

Evidence: “Experience with Windows and Java programming language.”

the obtained percentage of match would be 5/7 = 0.71, which is greater than 2/3 = 0.67 (the defined threshold value through which the evidence has been considered valid), and would result in a correct and presented evidence. However, the main concepts (the OS and programming language type) are different, and thus, it should not be that case.

Nevertheless, if we remove the articles and prepositions, the obtained percentage of match would be 3/5 = 0.6 (“Experience Windows Java programming language”), which is lower than 2/3 = 0.67, and would result in an incorrect evidence, and therefore, an omission of key evidence instance.

Figure 40 shows the implemented list of prepositions and other non-main words. This list can be updated if in future work more frequent words that should be eliminated are identified.

Comparing requirements with evidence

After that, and before comparing both of them, we check if the presented evidence is a justification or an evidence in itself. For this purpose, a list containing keywords or common expressions to detect rationales, such as “due to” or “because”, has been previously defined. We then check whether the current evidence contains some of those words. If that is the case, the “presented evidence” is a rationale and should not be considered as an omission of key evidence fallacy, even if the description does not match the requirement.

On the other hand, if the evidence does not contain any of the words of the justifications’ list, each word of the requirement is compared against each word of the evidence, and a percentage of match is calculated through divide the amount of matched words by the total number of words. After doing a series of test, we decided that if the percentage of match is greater than 2/3, the presented evidence match with the requirement, else, the presented evidence is considered as omitted. Figure 41 shows the code snippet that implements this functionality.

```
private String prepositions = " about , after , at , before , between , but , for , from , in " + " into , like , of , on , through , to , under , up , with " + " within , without , the , and , a , an , is , are ";
```

Figure 40. List of prepositions and articles

```
for (String word : words) {
    if (arraySkillsNoPrep[j].contains(word)&(percentMatch>=(double)1/3)) {
        usedSkills.add(j);
        resultsValidationStaff.add(true);
        isRationale = true;
        break;
    }
}
if (isRationale){
    break;
}else{
    if (percentMatch >= ((double)2/3)){
        usedSkills.add(j);
        resultsValidationStaff.add(true);
        j = arraySkillsNoPrep.length;
    }else{
        resultsValidationStaff.add(false);
        omittedReq.add(arrayRequirements[i]);
    }
}
```

Figure 41. Fallacy decision implementation
Finally, and after checking all the presented pieces of evidence against each requirement, we identify if there is any omitted evidence. If there is, it stores both the name of the element and its omitted evidence through the method shown in Figure 42. This method distinguishes the type of the received element, Role or Tool. The code shows the Role case, being the Tool case implementation analogous to this one. The goal of this operation is to store, in each position of a list, all the omitted pieces of evidence for the same Role. The name of this Role is stored in another list in the same position of the previous list. This will allow us to locate the omitted evidence for a Role when presenting the results and creating the validation report.

![Figure 42. Method to store the omitted evidence](image)

6.1.2. Fallacy detection plugin

In order to provide and automatise the functionality that prevents fallacies during the generation of process-based argumentations, all the concepts explained in Section 6.1.1 are implemented in an Eclipse platform plugin. In this way, it can be incorporated into the EPF Composer, which is included within the AMASS platform. The process models therefore can be validated whether they contain insufficient information that can cause the omission of key evidence fallacy in the process-based argumentations.

The plugin is invoked from a right-click menu as shown in Figure 43. The operation will only be applicable by clicking on those elements of type ProcessComponent (Capability Patter or Delivery Process) since these elements are the ones that have all the information of the modelled process.

![Figure 43. Fallacy detection plugin menu](image)

After invoking the plugin, a pop-up dialog appears to ask the user where he or she wants to store the results of the validation. For that, a Java method, specified in Figure 44, is implemented. In it we can see that a DirectoryDialog is created, where the system folders and directories will appear, allowing the user to navigate through them and select the target path for the validation reports.
Once the user has selected the target directory, two new “.txt” files are created under the specified folder, and the validation process starts. Besides, two “buffered writers” are created to collect the results (the omitted or sufficient information and the recommendations) while the fallacy detection algorithm is working. When this process finishes, the information that the buffers have gathered is used to both write the pertinent files and print it on the console. The information related to the Roles is stored under the file called “Staffing Plan Report.txt”, and the information regarding the Tools is shown in the “Tool Qualification Plan Report.txt” file. Moreover, this information is sorted in the following way: those elements with insufficient evidence appear in the first place, and then those whose information is sufficient are placed below. This structure will help the user since all the elements that need to be reviewed or fixed are presented in the first place.

```java
public void createDirectoryChooser() throws IOException{
    DirectoryDialog dialog = new DirectoryDialog(shell, SWT.Close);
    dialog.setMessage("Please specify a directory to store the result of the validation.");
    dialog.setText("Browse For Folder");
    String path = dialog.open();
    if (path != null) {
        FileWriter fileWriterStaff = new FileWriter(path + "/Staffing Plan Report.txt");
        BufferedWriter bwStaff = new BufferedWriter(fileWriterStaff);
        FileWriter fileWriterTool = new FileWriter(path + "/Tool Qualification Plan Report.txt");
        BufferedWriter bwTool = new BufferedWriter(fileWriterTool);
        }
else{
    confirmExit(2);
    }
}
```

Figure 44. Method to select the target directory of the report

Finally, the algorithm decides, based on the previous results, if the presented information in the process model is sufficient not to commit an omission of key evidence fallacy or if, to the contrary, there is a lack of critical information. In this last case, it will decide if the omission of evidence has occurred in the staffing plan, in the tool qualification plan, or in both of them. Then, a new dialog will appear to communicate the end of the validation process, and depending on the case, will allow either to open the specific file where the omission has been committed, or to directly open the folder containing both reports in the second case. This is implemented through a MessageDialog instance. To configure it with the proper information from the above-mentioned possibilities, two lists of Boolean variables store a true when the validation result is sufficient, and a false if not. Then, a method checks the contents of those lists to decide on which of the four possibilities we are (the presented evidence is sufficient, the presented evidence is insufficient in both requirements plans, the presented evidence is only insufficient in the tool qualification plan, and the presented evidence is only insufficient in the staffing plan). Figure 45 displays the appearance of the message dialog in this last case.

![Figure 45. Validation completion dialog](image)

### 6.2. Transform the validated process into a safety argumentation solution

After the validation process, and once all the required evidence is presented in the safety process, an automatic transformation of the validated process into a safety argumentation is performed. This is an essential function since the manual creation of safety arguments is a very laborious task and prone to errors. In this way, the implementation of an M2M transformation allows creating automatically a (validated) safety argumentation from a process model. This thesis takes an existing transformation, so Section 6.2.1 describes the main changes.
and updates performed to it. On the other hand, Section 6.2.2 explains the implementation of this transformation within an Eclipse plugin.

Figure 47 shows the main workflow of the transformation process. This includes some previous steps (marked on the diagram with a dotted line rectangle) that have to be done before launching the transformation plugin. These steps represent the functionality described in Section 6.1 and are needed to create a safety argumentation with no omission of key evidence fallacies.

6.2.1. Model to model transformation

As stated above, this work starts from an existing transformation written in ETL. The implemented changes are listed and explained below.

Addition of Role’s and Tool’s evidence

The previous implementation transformed the “process_lifecycle” plugin into the safety argumentation but did not consider the requirements of the standard and the evidence to comply with it. Therefore, the main change here is the introduction of the validated elements of the safety process. Specifically, the presented evidence—or the rationales for not including it—in the form of solutions and justifications.

In the previous version, and according to the MDSafeCer approach, each RoleDescriptor of the process was transformed into a Claim which stated that the Role should be certified. Then, a Solution which specified that the Role’s certification will be available. In the updated approach, the statement of the RoleDescriptor’s goal claims that the Role is certified, and then this is divided into as many Solutions as presented pieces of evidence. To do so, an operation called getSkills has been included, as shown in Figure 46. This operation receives the name of the RoleDescriptor that is under transformation and returns a sequence with its skills (evidence). For this, the first step is to obtain all RoleDescriptions involved in the process (line 2). Then, for each RoleDescription, we have to obtain the name of the Role that represents and check if it is the same that the one received in the operation (line 5).

It is important to mention that this name is encoded in the RoleDescription model as follows: name="Role_Name,Role_ID". Therefore, to obtain the name of the involved Role we have to get just the first part of the name field, i.e., from the zero position to the comma (line 4). In addition, the “cleaning” of the XML’s tags of the content is needed (lines 6 to 11), since to represent the argument in GSN, the statements of the nodes will contain the text included here.

Finally, and as it was done in the fallacy detection algorithm, we remove all the additional information such as the sentence that introduces the presented evidence. For this, first we split the skills by semicolons (line 14) to extract separately each piece of evidence. Then, we go through the first one, looking for certain words or symbols that introduce the presented evidence (if-else statement). Once this words or symbols are found, we keep the rest of the sentence (from that point until the end) as the first skill, and update the obtained skills with this new one.

```java
operation List<RoleDescriptor> getSkills(name: String): Sequence{
    var rolesDescriptions: Sequence = ROLESRoleDescription.allInstances();
    ...
    nameRole = n.name.substring(0, n.name.indexOf(",",));
    if (nameRole = name) {
        var sk0k0 = n.skills.replaceAll("\[\]<\[>\]",""");
        var sk0k1 = sk0k0.replaceAll("&nbsp;","");
        var sk0k2 = sk0k1.replaceAll("&",""");
        var sk0k3 = sk0k2.replaceAll(" expulsion",""");
        var sk0k4 = sk0k3.replaceAll(" &amp",""");
        sk0k5 = sk0k4.replaceAll("@x","");  
    }
    ...
    var splitsSkills = sk0k5.split(";");
    if (firstSkill.contains(";")){
        skill0 = firstSkill.substring((firstSkill.indexOf(";")+2));
    } else {
        if(firstSkill.contains("against")){
            var against: Sequence = firstSkill.split("against");
            skill0 = against.at(1);
        } else {
            if(firstSkill.contains("over")){
                var over: Sequence = firstSkill.split("over");
                skill0 = over.at(1);
            }
        }
    }
    return skills0;
}
```

Figure 46. Get skills operation
Figure 47. Flowchart of transformation plugin
The previous getSkills operation is invoked in the transformation for each RoleDescriptor (rd). Once the skills are obtained through the described operation, they have to be analysed in order to classify them either as evidence (solution) or rationale (justification). For this, for each of the obtained skills we created a Boolean variable that identifies if such skill is a justification or not. Then we look for the keywords that identify the rationales (because, due to, since), and if some of them are contained in the skill, an InformationElementCitation - Type#Justification element is created, and the justification variable is set to true. The name of the element will be Role_X's justification, and its description will contain the current skill in the following form: “Role_X's justification: Evidence 1 is not needed because...”. The id of the element will be defined by the number of the previous elements of the same type, and the id of the related RoleDescriptor, for example; id=32.R1234 would be the second Justification element for the Role whose id is 1234.

If the skill under analysis does not contain any rationales (justification variable equals to false), it will be transformed into an InformationElementCitation - Type#Solution. Similarly, the name of the element will be Role_X's certification, the description Role_X's certification against skill x, and the id E"n",R"id", where again, “n” is the number of the current evidence, and “id” is the identification of the associated RoleDescriptor. In addition, an AssertedContext is created to link the Claim with the Justification, and an AssertedEvidence to link the Claim with the Solution. Figure 48 shows the described transformation.

```
1 //Skills -> Solution/Justification
2 var skillsok = rd.getSkills(rd.name);
3 ...
4 for (sk in skillsok) {
5   if (sk.contains("because") or sk.contains("due to") or sk.contains("since")){
6     //Rationale -> Justification
7     iec.name = rd.name + "'s justification";
8     iec.id = rd.id + "-" + sk;
9     iec.description = rd.presentationName + "'s justification: " + sk;
10    iec.type = ARG!InformationElementType#Justification;
11    ...
12    justification = true;
13  }
14  }
15  if (justification <> true) {
16    //Skill -> Solution
17    iec.name = rd.name + "'s certification";
18    iec.id = rd.id + "-" + sk;
19    iec.description = rd.presentationName + "'s certification against " + sk;
20    iec.type = ARG!InformationElementType#Solution;
21    ...
22  }
```

Figure 48. Evidence transformation

The other elements validated in the process model are the Tool Mentors, so the update in the transformation is also performed. Analogous to the RoleDescriptors case, the previous transformation version included the transformation of these elements without its validation against the standard’s requirements. In this way, they were transformed into a Claim that stated Tool Mentor_X has to be qualified, and a Solution whose content was Tool Mentor_X’s qualifications. These qualifications are now obtained from the modelled process through the operation getToolQual, presented in Figure 49. In this case, before obtaining and “cleaning” the qualifications, we have to identify the Tool associated with the current ToolMentor. For this, we obtain all the Tools involved in the process (line 2) and its referenced ToolMentors. Then we compare the id for each of the ToolMentors with the current one, and if they match, it means that such Tool is the one that has the qualification (line 8). Once the related Tool is identified, we need to obtain the ContentDescription models, where the Key considerations field is modelled, and therefore the qualification.

This is carried out similarly to the previous case, with the name now encoded as name="Tool_Name,Tool_ID". Again, the XML tags are removed from the obtained key considerations, returning in this way the qualification of the Tool related to the ToolMentor. The part of the rule that transforms these qualifications into Justifications or Solutions is performed analogously to the previously described “Skills case” (Figure 48).
Laura Gómez Rodríguez

A tool-supported method for fallacies detection in process-based argumentation

Additional process elements

In the previous transformation, the argumentation starts in the Activity elements. Now, since the process is invoked from a DeliveryProcess, and the transformation follows the Work Breakdown structure, the top Claim included in the Case is the Capability Pattern process element. To implement this, the source element of the main rule is changed (line 3). In addition, the transformations of the new process elements are included. Figure 50 shows how the top-level Claim of the argument is derived from the Capability Pattern. From this element, we get the breakdownElements of type Phase (line 13). Then, an ArgumentReasoning element is implemented to connect the Capability Pattern with its Phases (lines 19 to 22).

Context

One key aspect in a safety argumentation is the concept of Context. This is important since it defines the scope of the process, and therefore the boundaries in which the argumentation is applicable. This concept has been
modelled in the EPF Composer processes under the Purpose field of the Delivery Process element. Therefore, to incorporate this concept we get the purpose from the content model of the Delivery Process (content.xmi) and after removing all the XML tags, we transform it into an InformationElementCitation – Type#Context as shown in Figure 51.

```java
1. var ctx = new ARG1InformationElementCitation;
2. var c = new ARG1AssertedContext;
3. // Purpose -> Context
4. var context = String = Lib2!DeliveryProcessDescription.allInstances().first().purpose;
5. var context2 = context.replaceAll(“\[\]”, “”);
6. var context3 = context2.replaceAll(“&#xD;”, “”);
7. var context4 = context3.replaceAll(“&#xA;”, “”);
8. var context5 = context4.replaceAll(“&lt;”, “”);
9. var context6 = context5.trim();
10. c.name = ph.name+"\’s context";
11. c.id = “C.Ph”+ph.id;
12. c.description = context6;
13. c.type = ARG1InformationElementCitationType#Context;
14. arg.argument.add(c);
```

**Figure 51. Context transformation**

### Undeveloped Work Products Claims

In a process-based argumentation, the earlier stages of the process such as the design and planning phase do not have any developed work product. These could be defined as expected outputs according to the standard with which the process shall be compliant. To implement this, leveraging the undeveloped symbol of GSN, we adding it to those Claims related to the Work Products belong to the planning phase of the process. This is implemented in ETL by setting the toBeSupported property to true (e.g. clw.toBeSupported = true).

#### 6.2.2. Transformation plugin

After updating the transformation code (.etl file), this functionality is implemented within an Eclipse plugin so that the transformation can be performed automatically. The previous implementation consisted of the transformation of the safety process, modelled with EPF Composer and exported in a single XML model which contained all the information. In the current implementation, there is no need to export the model to transform it. Because of the EPF migration, this can be done now from the Authoring perspective just by right-clicking on the desired process element to transform. Therefore, and as stated in Chapter 5, the EMF persistence implementation of the EPF Composer stores the container Method Library and all its contents elements in a repository of separated models (XMI files). Figure 52 shows an example of the structure of the generated model repository for a Method Library. In it we can see, for example, the three different XMI files that store the RoleDescription information for each Role (where the Skills field is included).

```
compliance modelling
| configurations
| mapping_requirements
process lifecycle
| standard_requirements
> project
| library.xmi
```

**Figure 52. Model repository structure**

Therefore, the first step for the class responsible for executing the transformation is to get the selected process element and the needed model files. Figure 53 shows the code snippet to obtain these elements. The method used to obtain the selected Delivery Process (lines 2 to 4) is the same that the explained for the fallacy detection case in Figure 36. After that, we obtain the single models’ files (the process_lifecycle’s plugin model in line 7, and
model and content files for the Delivery Process in lines 8 and 9), and the container folders for the multiple models (as the roles in line 10 and the tools in line 11). For these folders, we create two arrays with the list of all files that each folder contains (lines 13 and 14). Finally, the transformation file, i.e., the “.etl” file which contains all the rules and operations described in the previous section, is obtained in line 16.

```java
1 // Get the current delivery process
2 ISelection sel = page.getSelection();
3 TreeSelection tree = (TreeSelection) sel;
4 ProcessComponentImpl process = (ProcessComponentImpl) tree.getFirstElement();
5
6 // Get the needed model files
7 File plugin = new File(pluginPath); // ECP
8 File deliveryProcess = new File(directoryRootPlugin + "\deliveryprocesses\" + process.getName() + "/model.xml");
9 File deliveryProcess2 = new File(directoryRootPlugin + "\deliveryprocesses\" + process.getName() + "/content.xml");
10 File directoryRoles = new File(directoryRootPlugin + "/roles"); // ROL
11 File directoryTools = new File(directoryRootPlugin + "/tools"); // TOOL
12 String[] listRoles = directoryRoles.list();
13 String[] listTools = directoryTools.list();
14 File etlFile = new File (pluginPath + "/epsilon/epflargumentation.etl");
```

Figure 53. Obtaining the models

After getting the needed files, we have to create the corresponding EMF models. For that, two private methods have been adapted from the previous plugin. Both methods responsible for creating the source (createEMFSourceModel) and target (createEMFTargetModel) models have the following variables:

- **String name**: The name of the model. This is the one used in the transformation file to refer the elements belonging to this model.

- **String metaModelFilePath**: The path of the file that contains the proper metamodel.

- **String modelFilePath**: The path of the file that contains the model. In the case of the source models (those obtained in Figure 53), the path where they are stored, in case of the target one, the path where we want to save it.

- **Boolean readOnLoad**: This Boolean variable indicates if the model has to be read before performing the transformation. If true, the model will be read previously, accessing to the model information. Therefore, this should be true for the source models (since they have the information), and false for the target one (since it is empty when is created).

- **Boolean storeOnDisposal**: This Boolean variable indicates if the model has to be stored after performing the transformation. If true, the model will be saved with the changes applied by the transformation; if false, the changes will be discarded after disposal. Therefore, this should be true for the target model (since it will contain the transformed model that we want to obtain) and false for the source ones (since they already have been read and used and do not experience any changes).

In addition, the method responsible for creating the target model (createEMFTargetModel) receives the previous variables, plus one additional variable: **String targetMetaModelURI**, which specify the URI of the registered metamodel (“arg” for this case).

Figure 54 shows the creation of both the source and the target models. The case of the Roles and Tools models needs to be remarked since they have as many files as elements involved in the safety-process. For these, we go through each list previously obtained (which contain all the model files) and create a different EMF model for each element. These models have the same name of the element they represent (since it must be unique), but a common alias for those of the same type (“ROL” for Roles and “TOOL” for Tools). The aliases allow forming groups of models inside the repository to perform aggregate operations on those models under the same alias. Therefore, since the number of Roles and Tools involved in each safety-process may vary, these EMF models have to be created dynamically and grouped under a common alias. For these cases, the transformation file refers to the alias instead of to the model name to get the containing elements.
Finally, the created EMF models are added into the ETL module and this is executed to perform the transformation. Once this finishes, the target model is saved locally in a new project into the current Workspace under the name Argumentation.

Since this functionality is implemented within AMASS platform, which includes OpenCert as the main tool for management of assurance and compliance, we need to adapt our solution to the requirements that OpenCert presents. One of them is the use of the CDO Repository to persist the generated model and the associated diagram. This CDO Repository is composed of Assurance Projects, which allow maintaining the life-cycle of projects. Such Assurance Projects are in turn composed of four folders (ASSURANCE_PROJECT, EVIDENCE, ARGUMENTATION, and PROCESSES) to store the different type of elements. Therefore, as the generated model and diagram represent the safety argument, we will store both files in the ARGUMENTATION folder.

For all the above, the first step of the transformation process is to select the previously created Assurance Project folder in which the user wants to store the results. To do so, firstly we need to initiate a connection with the CDO Repository and create a transaction which will commit all the changes done. This connection is established by using the parameters that the user has previously configured in the Eclipse preferences, which included the name of the repository, the used protocol, and the server name. Once the connection is established, a dialog which shows all the Assurance Projects in the CDO repository is created through a GUI class which has been implemented in the plugin. Finally, the chosen Assurance folder name is stored in the selection variable, so we can determine where we have to store the results. Figure 55 shows the details of this implementation.

```java
1 //Connect to the CDO Repository
2 CDOConnectionUtil.instance.init(
3     PreferenceConstants.getRepositoryName(),
4     PreferenceConstants.getProtocol(),
5     PreferenceConstants.getServerName());
6 CDoSession session = CDOConnectionUtil.instance.getCurrentSession();
7 CDoView view = session.openView();
8 CDoTransaction transaction = session.openTransaction();
9
10 GUI gui = new GUI(shell,view);
11 gui.create();
12 gui.open();
13
14 String selection = gui.resourceName();
```

Figure 55. Connection to the CDO Repository

Once the target Assurance Project folder is selected, the previously explained transformation process is executed. When the transformation finishes and the generated model is stored in the local project folder, a copy of this is uploaded to the CDO Repository as a CDOResource. Figure 56 shows the method used to implement this, which was obtained from the previous implementation and updated to the needs of the current one. To do so, first,

---

1 See https://www.polarsys.org/opencert/resources/gettingstarted/
Laura Gómez Rodríguez  

A tool-supported method for fallacies detection in process-based argumentation

an empty CDOResource is created in the ARGUMENTATION folder of the selected Assurance Project under the name “argModel.arg” (line 2). Then, the generated local model file is obtained to derive the elements it contains. To do so, we create a new empty Resource from the URI of such model file (line 7), and through a FileInputStream, we load all the contents into the created Resource. Finally, we load all the contents of this Resource into the empty CDOResource (line 14) and save the updates.

```java
1 //Create the model CDO Resource
2 CDOResource argModel = transaction.getResource(“/"+selection+"/ARGUMENTATION/argModel.org”);
3 String argModelFile = ResourcesPlugin.getWorkspace()
4 .getResource().getLocation().toString()+”/Argumentation/argModel.org”;
5 ...
6 Resource argXMLResource = resourceSet.createResource(argXMLUri);
7 File newFile = new File(argModelFile);
8 FileInputStream argFileInStream = new FileInputStream(newFile);
9 argXMLResource.load(argFileInStream,Collections.EMPTY_MAP);
10 CDOResource argCDOResource = transaction.getResource(“/"+Selection+"/ARGUMENTATION/argModel.org”);
11 argCDOResource.getContents().addAll(argXMLResource.getContents());
12 ...
13 argCDOResource.save(Collections.EMPTY_MAP);
```

Figure 56. Load model into CDO Repository

After that, the diagram file needed to represent the argumentation in the OpenCert assurance case editor is generated by a method called createArgDiagram(). This method was adapted from the class org.eclipse.opencert.sam.arg.arg.diagram.part.ArgNewDiagramFileWizard.java included in the OpenCert source code. This method generates a diagram from a model file selected by the user in a wizard. The main difference that has been introduced in this method (presented in Figure 57) is that now the diagram is always generated from the local model file (line 3), and is stored in the same location (line 6), removing the wizard selection.

```java
1 public void createArgDiagram(){
2     IPath workspace = ResourcesPlugin.getWorkspace().getRoot().getLocation();
3     IURI argModelURI = URI.createPlatformResourceURI(”Argumentation/argModel.org”, true);
4     LinkedList<IFile> affectedFiles = new LinkedList<IFile>();
5     File file = new File(workspace,”/Argumentation/argDiagram.org_diagram”);
6     try {
7         file.createNewFile();
8     } catch (IOException e) {
9         e.printStackTrace();
10     }
11     IPath locationPath = Path.fromOSString(file.getPath());
12     IFile diagramFile = ResourcesPlugin.getWorkspace().getRoot().getFileForLocation(locationPath);
13     ArgDiagramEditorUtil.setCharSet(diagramFile);  
14     affectedFiles = findAffectedFiles(diagramFile);
15     URL diagramModelURI = URI.createPlatformResourceURI(”Argumentation/argDiagram.arg_model”, true);
```

Figure 57. Create diagram method

Once the diagram has been generated locally, the same procedure explained in Figure 56 is followed to load this diagram file into the CDO Repository. Then, the dialog shown in Figure 58 will appear to inform the user that the transformation process has finished successfully, listing the locations where the generated files have been stored.

```
Transformation completed

The model and diagram are generated under
- ARGUMENTATION folder of the project Test_Assurance_1 in the CDO Repository
- Argumentation project in the current workspace

OK
```

Figure 58. Transformation completed dialog
Finally, when the user clicks on the above “OK” button, the process is finished by opening the diagram editor in the resource perspective of the Eclipse workbench. This perspective shows the created project which contains the locally generated model and diagram files. This functionality has been implemented through the Java code shown in Figure 59.

```java
// Change the perspective
if (PlatformUI.getWorkbench() != null) {
    IPerspectiveDescriptor descriptor = window.getWorkbench()
        .getPerspectiveRegistry().findPerspectiveWithId("org.eclipse.ui.resourcePerspective");
    PlatformUI.getWorkbench().getActiveWorkbenchWindow().getActivePage().setPerspective(descriptor);
} else {
    MessageDialog.openError(shell, "Selection error", "An assurance project must be selected in order to
open the diagram

    try {
    ArgDiagramEditorUtil.openDiagram(diagramResource);
    } catch (PartInitException ex) {
    ArgDiagramEditorPlugin.getInstance().logError(
    "Unable to open editor", ex); // $NON-NLS-1$
    }
```

Figure 59. Open diagram
7. Case study

Following the methodology proposed in Chapter 5, Section 1, and after developing the proposed solution detailed in Chapter 6, the application of the overall approach is illustrated through a case study. This is based on the process related to the software development associated with AOCS (Attitude and Orbit Control Subsystem), following the requirements from the ECSS-E-ST-40C standard. This chapter is structured as follows: Section 7.1 presents the followed standard, in addition to the extracted requirements. Section 7.2 describes how the process is modelled in EPF Composer. Sections 7.3 and 7.4 show the results obtained by performing the fallacy detection and the transformation on the modelled process, respectively. Finally, Section 7.5 discusses the obtained results.

7.1. Description

AOCS mainly provides to the satellite the means to orient its optical reference axis to the Sun (through the Attitude Control), and to control the position of the satellite in orbit, controlling the roll angle performing orbital manoeuvres (through the Orbit Control) [53], [54]. This is used in a number of different telecommunication satellite platforms [54], so the safety of this subsystem in its intended environment must be assured, both the software and the hardware elements. Regarding the software ones, in any European space project the development of any software must be, according to [54], fully compliant to at least the following standards: ECSS-E-ST-40C Software general requirements, and ECSS-Q-ST-80C Software product assurance. In addition, for the software related to safety-critical applications, such as the AOCS one, a more thorough process of assurance is needed. This includes a series of activities to fulfill the standard’s requirements as well as the provision of the required evidence to prove its compliance.

For all of the above, this case study is based on the ECSS-E-ST-40C standard in order to comply with the safety-critical AOCS software development process, specifically on Clause 5.5 (Software Design and Implementation Engineering Process).

ECSS-E-ST-40C is one of the standards of the European Cooperation for Space Standardization (ECSS) series, which are “intended to be applied together for the management, engineering and product assurance in space projects and applications” [6]. Specifically, this standard concerns the space software engineering processes and covers all aspects related to them, such as the requirements definition and the design. Clause 5 defines the requirements for any space project producing software for space systems. These requirements are expressed in the following structure: for each one, the requirement is identified by four numbers (according to the clause they belong), followed by a letter if several requirements are needed. After that, the EXPECTED OUTPUT section provides the associated outputs for that requirement, identified by different letters when more than one outputs are expected. Figure 60 shows an example of this.

Clause 5.5 states that the software design and implementation engineering process consists of three phases: design of software items, coding and testing, and integration. Each of these phases is composed of different activities (e.g., detailed design of each software component), and they in turn are separated in several tasks (e.g., develop a detailed design for each component of the software and document it). Following the structure detailed above, Figure 60 shows the requirements of the three tasks for the first activity of the first phase. All the requirements of Clause 5.5 were extracted and used to model the compliant safety life-cycle process in EPF Composer, which is detailed in Section 7.2.

In addition, to ensure the quality and safety of a process, as important as the process followed is the qualification of the roles and tools involved to develop it. However, ECSS-E-ST-40C standard does not specify explicitly requirements related to skills or key competencies of roles. To model these, the EN 50128 standard (railway domain), as well as the industry requirements, were analysed in order to extract and adapt the needed skills for roles.

Specifically, as Gallina et al. derived in [55], according to Table B.2 of the EN 50128 standard’s Annex B, “a designer shall: transform specified software requirements into acceptable solutions, own the architecture and downstream solutions, define or select the design methods and supporting tools, apply appropriate design principles and standards, develop component specifications where appropriate, maintain traceability to and from the specified software requirements, develop and maintain the design documentation, and ensure design documents are under change and configuration control. In addition, with respect to expected competencies a designer shall be competent in: engineering appropriate to the application area, the safety design principles, design analysis & design test methodologies, understanding the problem domain. Moreover, a designer shall understand: the constraints imposed by the hardware platform, the operating system and the interfacing systems and the relevant parts of EN 50128. Finally, (s)he shall be able to work within design constraints in a given environment.”
From these requirements and the industry desired ones, the adapted key competencies for the AOCS Software Architect role for our specific case are as follows:

An AOCS SW Architect shall be competent and have relevant experience in: understanding the problem domain, design analysis and design test methodologies, perform planning activities, understanding all the constraints imposed by the hardware platform, the operating system interfacing systems, working experience within design constraints in a given environment, knowledge of AOCS/GNC system design & real-time implementation, knowledge of Matlab/Simulink tools for modelling and functional simulation, and university degree in engineering.

### 5.5.2 Design of software items

#### 5.5.2.1 Detailed design of each software component

- a. The supplier shall develop a detailed design for each component of the software and document it.
  
  **EXPECTED OUTPUT:** Software components design documents [DDF, SDD; CDR].

- b. Each software component shall be refined into lower levels containing software units that can be coded, compiled, and tested.
  
  **EXPECTED OUTPUT:** Software components design documents [DDF, SDD; CDR].

- c. It shall be ensured that all the software requirements are allocated from the software components to software units.
  
  **EXPECTED OUTPUT:** Software components design documents [DDF, SDD; CDR].

Figure 60. “Design of software items” requirements

The required skills for the other roles involved in the process (AOCS AIT –Assembly Integration and Test– Engineer, AOCS Engineer, AOCS Software Verification and Validation Manager, and Development Team Leader) were derived from the industrial requirements from positions for AOCS software development and are summarised in Table 6.

<table>
<thead>
<tr>
<th>ROLE</th>
<th>REQUIRED SKILLS</th>
</tr>
</thead>
</table>
| AOCS AIT                  | Engineering degree (aviation and/or flight control), experience in Attitude & Orbit
  Control Systems (AOCS) and Failure Detection & Recovery (FDIR) verification tests, knowledge of European Cooperation for Space Standardization (ECSS) is desired, experience with Linux. |
| AOCS Engineer             | University degree in engineering, several years of experience in the design, analysis and simulation of AOCS systems in different project phases, excellent hardware knowledge (sensors, computer, actuators), experience in AOCS AIT at subsystem and system level, and working experience with Linux System, Matlab and SatSim. |
| AOCS SW V&V Manager       | AOCS SW planning and test methodologies, experience in AOCS SW V&V activities and techniques, able to support the Electra AOCS SW V&V Team, knowledge of AOCS system design, knowledge of safety attributes of the applications domain, good analytical and problem-solving skills, university degree in engineering. |
| Development Team Leader   | Management of Electra AOCS SW Development Team, working experience with Matlab/Simulink, knowledge of design analysis and design test methodologies, good analytical and problem-solving skills. |
7.2. Process modelling in EPF Composer

After extracting all the requirements and recommendations from Clause 5.5 of the ECSS-ST-E-40C standard regarding the planning process, and deriving those related to skills and key competencies from the EN 50128 standard and industry requirements, the modelling in EPF Composer is performed. To do so, the method described in Chapter 5, Section 2.1 has been followed. Therefore, after creating a compliance_modelling plugin to customise the icon of the practice associated to the requirements (see Figure 27), the derived requirements are modelled as practices extending this ‘requirement’ practice, in a plugin called ecss-e-st-40c_requirements as shown Figure 61.

![Figure 61. Modelling ECSS-E-ST-40C standard requirements](image)

Besides, the figure above shows the requirements related to the three main phases (design of software items, coding and testing, and integration) modelled as Practices, and their related activities as sub-practices. The practices marked with a blue and an orange rectangle are the ones specified in the example of Figure 60. In addition, marked with a red rectangle are those requirements related to the omission of key evidence fallacy, that is, the staffing plan and the tool qualification plan ones. Figure 62 shows an example of how the extracted requirements for the Roles (presented in Table 6) are modelled following the method proposed in Chapter 5, Section 2.1, that is, in the Brief description field of the extended Practice, and separated by semicolons. Figure 63 shows an example of the Tools.

![Figure 62. Modelled requirements for AOCS engineer Role](image)
Once all requirements have been extracted and modelled, the associated life-cycle is modelled in another plugin called `process_lifecycle`. Figure 64 shows the Method Content elements, grouped by the three phases defined in the standard. Within each package, all the associated Tasks, Work Products and Guidances are included. Blue and green rectangles highlight those requirements shown in Figure 60. Besides, an additional Content Package called `organization` is modelled to gather all the Roles involved in the process.
In addition, the certifications related to such Roles, as well as those qualifications regarding the involved Tools, shall be modelled as presented evidence following the guidelines proposed in Chapter 5, Section 2.1. Figures 65 and 66 show an example of the modelled evidence in the rich text editor of EPF Composer for a Role and a Tool, respectively.

**Figure 65. Modelled evidence for AOCS engineer Role**

**Figure 66. Modelled evidence for DOORS Tool**

On the other hand, the Process elements, sequenced in compliance with Clause 5.5 of the ECSS-E-ST-40C standard have been modelled in the Process package as Figure 67 shows. Again, the process elements circled with coloured rectangles refers to those presented in Figure 60. Below Work Breakdown Structure, the involved Roles through their associated Role Uses are shown.

**Figure 67. process_lifecycle process elements**
7.3. Omission of key evidence fallacy detection application

Once all the requirements and the process for AOCS software development have been modelled as method plugins in EPF Composer (as shown previous section), the detection of the omission of key evidence fallacy is performed on the process model. This application is invoked from a contextual menu depicted in Figure 43, either on the Capability Pattern or on the Delivery Process of the modelled process. Then, after specifying the target directory to store the result of the validation, the message shown in Figure 68 is obtained. The dialog contains the list of those elements with insufficient information to fulfil the modelled requirements. In this case we can see that both the Staffing plan and the Tool qualification plan present elements with insufficient evidence, specifically Roles AOCS Engineer and Development Team Leader for the former, and Matlab and Simulink Tool Suite Tool for the latter.

![Figure 68. Insufficient evidence dialog](image)

The details of the results such as the particular omitted evidence and the recommendations, as well as the list of the elements containing sufficient information are presented both in the validation reports (stored in the target folder previously selected) and printed on the console. Figure 69 shows the content of the Staffing Plan report, that is, the results for the detection of fallacies of the Roles. In the first place, and enclosed in a red box, appears the Roles listed in the previous dialog, i.e., those with insufficient evidence. In addition, the specific omitted certifications are listed under the DETECTED FALLACIES label, and the proper actions to take in order to meet the requirements, under the RECOMMENDATION label. In the second place, and marked with a green box, the Roles with sufficient evidence are listed. Figure 70 shows the obtained results printed on the console for the Tool Qualification plan, structured in the same way as the just presented for the Staffing plan.

![Figure 69. Staffing Plan report](image)
Looking at either the fallacy reports or the console we can see that the presented skills certifications for Roles AOCS AIT, AOCS SW Architect, and AOCS SW V&V Manager, and the presented tool qualifications for Tools DOORS, Eclipse, and SPARC-RTEMS-GCC are sufficient. Nevertheless, the AOCS Engineer and Development Team Leader Roles, and the Matlab and Simulink Tool Suite Tool have omitted key evidence. Particularly, AOCS Engineer has omitted two pieces of evidence (university degree in engineering, and working experience with Linux System, Matlab and Satism), while for the Development Team Leader no evidence has been presented.

By following the recommendations, Table 7 shows the skill certifications—and rationales for the omission—that have been added. For the Development Team Leader case, the added evidence is expressed exactly as the requirements. However, for the AOCS Engineer the order and content of the presented evidence have been changed—maintaining the same meaning—to check the developed match algorithm works correctly. In the case of the Matlab and Simulink Tool Suite, a rationale for the omission of the evidence is provided.

<table>
<thead>
<tr>
<th>Element</th>
<th>Evidence / Rationale</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development Team Leader</td>
<td>Development Team Leader certification against competency in management of Electra AOCS SW Development Team; working experience with Matlab/Simulink; knowledge of design analysis and design test methodologies; good analytical and problem-solving skills.</td>
<td>Shall be competent and have relevant experience in: Management of Electra AOCS SW Development Team, working experience with Matlab/Simulink, knowledge of design analysis and design test methodologies, good analytical and problem-solving skills.</td>
</tr>
<tr>
<td>AOCS Engineer</td>
<td>Degree in engineering from the Mälardalen University</td>
<td>University degree in engineering</td>
</tr>
<tr>
<td>AOCS Engineer</td>
<td>Experience in working with Matlab, Satism and Linux</td>
<td>Working experience with Linux System, Matlab and Satism</td>
</tr>
<tr>
<td>Matlab and Simulink Tool Suite</td>
<td>Qualification not needed since source code is fully tested</td>
<td>Matlab and Simulink Tool Suite shall be qualified</td>
</tr>
</tbody>
</table>

Finally, Figure 71 shows the dialog obtained once all the previous evidences have been added to the corresponding method elements, and all the requirements have been fully satisfied. In this way, we already can transform the process model into a safety argument without any omission of key evidence fallacy instance.
7.4. Transformation application

As the flowchart of Figure 47 details, before performing the transformation, all required evidence must be provided to avoid omission of key evidence fallacies when the argumentation is created. Therefore, once all the detected fallacies have been eliminated by modifying and remodelling the process, and the fallacy detection plugin returns the sufficient evidence dialog (shown in Figure 71), the automatic generation of the safety argumentation is performed. This application is invoked in the same way that the fallacy detection one but, through the menu called “Transformation”, and the “EPF2Argumentation” submenu.

Since, as discussed in Chapter 6 Section 2.2, the generated model and diagram have to be stored into the CDO Repository within an Assurance Project, the first step before performing the transformation is the creation of a new Assurance Project. This is done by following Chapter 5.1 “Create Assurance Project and Baseline” of [56]. Once this is created, and the connection settings have been configured properly, the EPF2Argumentation plugin is invoked. For the sake of clarity in the visualization of the results, Figure 72 shows part of the model and the outline of the obtained argument including only one phase; Integration. Nevertheless, the generated model and diagram contains all the elements modelled in the process.

![Figure 72. Generated model and diagram](image)

The generated model and diagram, compliant with SACM metamodel, are rendered via GSN and visualised in the OpenCert assurance case editor. Figures below show the parts of the argumentation in which this thesis has focused on. Specifically, Figure 73 shows the context in which the process is based, that is, the ECSS-E- ST-40C standard. Figure 74 shows the rationale included in the previous section (Table 7) for the justification of the omission of Matlab and Simulink Tool Suite qualification, as well as the undeveloped goal for the Work Product of this planning phase of the process. Finally, Figure 75 shows the sub-claim that states the AOCS Engineer certification. This sub-claim is solved by the five presented pieces of evidence against the requirements for that role. Only the properties (i.e., the Id, Description and Type) for the last solution are displayed, but the information for the rest of the pieces of evidence can be obtained by clicking on each element.

![Figure 73. Representation of the context in the generated argument](image)
Figure 74. Representation of Work Products and some Guidance elements in the generated argument

Figure 75. Representation of the AOCS Engineer role in the generated argument
7.5. Discussion

In this chapter we have presented the results of applying the overall approach of our solution to a particular and real case. This solution is based on the Model-Driven Safety Certification (MDSafeCer) method [11], which includes the following steps:

1. Safety-process modelling
2. Process-based argument generation
3. Process-based argument Check&Completion

However, in the approach presented in this thesis, an additional check step is performed before generating the process-based argument, i.e., in the safety-process model, through the fallacy detection solution described in Chapter 6, Section 1. This task is performed iteratively until all the required evidence has been presented and the result of the validation is satisfactory. This functionality allows the generation of the argumentation only once it has been validated, that is, once has been checked that there is no missing or wrong information in the process model. In addition, since this validation has been implemented to perform it automatically, a reduction in the time, cost and effort of check arguments is expected to be obtained. For verify that, an additional experimental validation would be needed. For example, by writing a piece of code we can be able to check how much time is taken for loading elements, fallacy detection and transformation. Also, an experiment for determining the gain and the reduction of time by measuring the time difference between manually creating and review arguments and automatically could be performed.

From the case study, it was observed that the proposed algorithm to detect the omission of key evidence fallacy works correctly, detecting adequately both the omitted and the incorrect evidence. The distinction between the evidence in itself (solution) and the rationale for not providing it (justification) is also correct.

Then, once the process has been modelled adequately, and all the required information to comply with the standard has been provided, the process-based argument is generated. This is also performed automatically, which reduces significantly the time of creation of the safety argumentation, the required knowledge to generate it, and the possible errors that a human can commit in such task.

Therefore, the steps followed in this solution are the listed below, iterating, as said above, the steps 1 and 2 until the adequate process model is obtained. The concept and design of this approach have been accepted in the 11th International Conference on the Quality of Information and Communications Technology (QUATIC) [57].

1. Safety-process (re)modelling
2. Fallacy detection
3. Process-based argument generation

The application of the solution presented in this thesis allows to perform automatically some mechanical tasks and to avoid unnecessary repetitive work, expecting thus the increase in effectiveness in terms of time, effort and cost. In addition, the automatic detection of fallacies assures that the generated argument contains the sufficient evidence to fulfil the standard’s requirements, improving thereby the quality of the generated argumentation.

Hence, through this discussion, we can answer –justifiably– the third research question raised in Chapter 4; “Can we increase the quality of safety arguments?” in an affirmative way.
8. Conclusions

This chapter presents the overview of the work performed in this thesis. Specifically, Section 8.1 summarises the purpose, issue and main results of the work. Section 8.2 identifies the limitations founded during the development of the thesis and proposes future lines of work to improve it.

8.1. Summary

This thesis has studied and analysed the safety arguments within safety cases. Specifically, the process-based arguments, that is, those safety arguments that aim to ensure and justify that the process followed during the development of a system is compliant with some standard. Standards such as ECSS-E-ST provide guidance on the steps and processes that must be followed to develop a safety-critical system (e.g. AOCS). As stated in Chapter 2, compliance of processes is mandatory for certification purposes, so the generation of such process-based arguments is essential, but also a costly and time-consuming task. In addition, a proper reasoning as well as presenting sufficient and adequate evidence to support it is needed. Therefore, the automatic generation of fallacy-free process-based arguments would help to alleviate this process and solve this issue.

This challenge was addressed by this thesis through the proposal of a solution composed of two methods; the automatic detection of fallacies, and the automatic generation of process-based argumentation.

For the first one, described in Chapter 6, Section 1, only the detection of the omission of key evidence fallacy has been implemented. This occurs when an argument fails to provide one or more of the expected evidences—in number or type—to support its particular claims. In this type of arguments (process-based), the expected evidence is associated to the certification, qualification, and provision of the involved elements, that is, the roles that perform the process, the tools they use to do so, or the documents that support their tasks such as guidelines or examples. Therefore, the fallacy detection method implemented in this thesis checks if all the required evidence is presented in the process, modelled in compliance with SPEM 2.0/UMA through the EPF Composer modelling tool. To do so, both the requirements and the process life-cycle was modelled following the best practices proposed by IBM. However, since EPF Composer does not support the definition of user-defined types, the guidance type Practice for requirement was customised with an icon and variability relationships as described in [59].

For the second part of the solution, presented in Chapter 6, Section 2, the automatic generation of the process-based argument from the process model was performed through Eclipse Modelling Framework (EMF) and Epsilon Transformation Language (ETL). Then, the obtained model and diagram files are persisted into the CDO repository of OpenCert and visualised into its assurance editor, rendered via GSN.

As illustrated in Chapter 7, the validation of this solutions has been performed through a case study based on the AOCS software development process in compliance with ECSS-E-ST-40C standard. Since key competencies requirements for roles are no expressed in this standard, they have been extracted from EN 50128 standard and industry requirements. The application of the proposed methods to this case study has revealed the adequate working of the solution as well as the expected major benefits that it would provide in terms of time, cost, effort, and quality.

8.2. Future work

During the development of this thesis, some limitations have been found, which are listed below:

- The proposed solution only focuses on one of the several [21] types of fallacies that can be found in safety arguments. Due to the time limitations of the thesis, only the omission of key evidence fallacy has been analysed, so that, although this improves the quality and the correctness of the generated argument, there may be other types of fallacies in it.

- In addition, in the context of this fallacy, the process-based arguments shall provide related and relevant evidence to support crucial process elements. In this thesis, only the two main elements have been implemented (Roles and Tools), leaving others such as Guidelines or Examples to be implemented.

- The Capability Patterns (Process Patterns) presented in the modelled processes are described in the SPEM 2.0 documentation [8] as “reusable building blocks for creating new development processes”. For this purpose, GSN provides the Patterns [18, pp. 15-17], so these process elements could be transformed into such GSN Patterns. Due to time limitations, this functionality has not been added to the transformation.

From these limitations and with the aim of improving the current implementation, the following future lines of work have been derived:
• In order to improve the quality of the safety argument, the validation of other –automatically detectable– fallacies that commonly appears in safety arguments could be implemented. To do so, a deeper analysis of how the remaining fallacies can be automatically detected should be carried out.

• The implementation to detect the omission of key evidence fallacy of the remaining available elements –on the planning phase of the development processes– should be performed. For example, the Guidelines or Examples elements could be validated if they include an attached file, description or URL as evidence that such elements are provided.

• Although different words and expressions to detect evidences and justifications have been defined, a research on different safety arguments could be performed in order to identify patterns of phrases, words, and expressions commonly used in them, refining thereby the current lists. The improvement of the match algorithm can also include a “library” of synonyms which allows it to identify texts with the same meaning even if the percentage of match is not enough to be considered as correct evidence.

• Due to the modular nature of the safety cases and the development processes, the possibility of reusing certain common structures or parts of the process-based arguments through encapsulating it in Patterns could be added. Leveraging the reusable purpose that Process Patterns (Capability Patterns) of SPEM 2.0 provide, they could be represented via these modelling elements. This not only reduces the time and complexity to develop the arguments but also improves the quality of them since these structures have been previously validated, so the use of such patterns ensures the correctness of the argument. This modularity can also be improved by adding a contract-based approach to organise and join the tasks performed by different teams working in different departments, and therefore, their different process-based argument fragments [11].

• In order to validate quantitatively the proposed solution, experimental validation in addition to metrics to measure the reduction of time would be needed.
References


Appendix A: User Manual

This appendix presents the steps that the user needs to perform in order to use the proposed solution. We assume that the process has already been modelled with EPF Composer. The guidance for the creation of process elements is provided in Chapter 2, Section 1, and also available in the EPF Composer User Manual [58].

1. Firstly, the Fallacy Detection plugin has to be launched. In this context, open the Authoring perspective following Window > Perspective > Open Perspective > Other… > Authoring as shown in Figure A.1. In that perspective, open the Method Library modelled with EPF Composer, as Figure A.2 shows.

Figure A. 1. Open Authoring Perspective

Figure A. 2. Open Method Library
2. Once the Method Library is opened in the selected workspace then right-click on the process to validate it, either on its Capability Pattern or on its Delivery Process. Then the “Fallacy Detection” menu, “Detect Omission of Key Evidence” submenu need to be pressed, as is shown in Figure A.3.

![Figure A. 3. Fallacy Detection plugin menu](image)

3. When the plugin starts, the dialog shown in Figure A.4 will appear asking the user to specify a target directory to store the results of the validation. Then, by clicking on the “Browse” button, a file chooser appears to select the desired folder or directory, where the validation reports will be stored. Figure A.5 shows an example of this dialog.

![Figure A. 4. Select directory dialog](image)

![Figure A. 5. Directory dialog](image)
4. After selecting the target directory, the validation process starts. When the fallacies detection finishes, four different messages can be obtained, depending on the number and type of the detected fallacies. If the algorithm has detected fallacies only in either the staffing plan or the tool qualification plan, the dialog allows the user to open the related report by clicking the button “Open File”. Otherwise, the user can open the target folder by clicking the button “Open Folder”. Figures A.6 and A.7 show an example of these messages. Besides, the validation results are printed on the console as Figure A.8 shows.

![Validation Completion dialog - Open File](image)

**Figure A.6. Validation Completion dialog - Open File**

![Validation Completion dialog - Open Folder](image)

**Figure A.7. Validation Completion dialog - Open Folder**

![Console results](image)

**Figure A.8. Console results**

5. Once the *Fallacy Detection* plugin reports that the presented key evidences are sufficient (see Figure A.7), the transformation plugin can be run. For that, the properties of the repository have to be properly configured and the creation of an *Assurance Project* to store the results of the transformation is needed. To configure the repository, the user has to click Window > Preferences. Then, in the Preferences window, open the “Opencert” label on the left-side menu, click on “Model Repository Preferences”, and fill the fields with the values shown in Figure A.9
To create an Assurance Project click on File > New > Other... Next, in the wizard selector, look for OpenCert folder, open it, select “New Assurance Project” and follow the instructions. Figure A.10 shows this, and Chapter 5.1 “Create Assurance Project and Baseline” of [56] specifies the steps needed to create the project. If the user has already an Assurance Project this step can be skipped.

6. After obtaining the sufficient information message and creating the Assurance Project the transformation plugin can be run. For that, right-click on the Delivery Process of the process to transform, and then on the “Transformation” menu, “EPF2Argumentation” submenu, as shown Figure A.11.

---

Figure A. 9. Model Repository Preferences

Figure A. 10. Create a new Assurance Project
7. Once the transformation plugin is invoked, the dialog shown in Figure A.12 will appear asking the user a target Assurance Project from the CDO Repository. By clicking on the “Browse…” button, the window shown in Figure A.13 appears to navigate and select the previously created Assurance Project folder to store the results of the transformation, i.e., the generated model and diagram.

Figure A. 11. Transformation plugin menu

Figure A. 12. Select Assurance Project dialog

Figure A. 13. Assurance Project selection window
8. After selecting the Assurance Project folder, the transformation process starts, and a progress information monitor as the one shown in Figure A.14 will appear to inform the user about the tasks that the plugin is performing.

![Figure A.14. Progress Information message](image)

9. After completing the transformation process, a new dialog will appear to inform the user where the generated results have been stored. Figure A.15 shows an example of this message.

![Figure A.15. Transformation completed message](image)

10. By clicking on the “OK” button, the progress information monitor shown in Figure A.16 appears to inform the user that the diagram is opening. Finally, when this task finishes, the generated diagram is opened into the Opencert assurance case editor, as Figure A.17 shows. In addition, we can open the local model, and both the model and diagram stored in the CDO repository. The local diagram and model are stored in a folder called Argumentation, created under a new project into the user workspace, and can be found in the Project Explorer tab, highlighted in green in Figure A.17. These files are also stored into the CDO repository, highlighted in red in Figure A.17. To open the repository explorer it has to be clicked Window > Show View > Other… and under the Opencert folder click on Repository Explorer. To visualise the diagram stored in the CDO repository the user needs to double click on it, and the diagram will be opened in the editor window. Then the user has to select the elements from the Navigation View, drag them and drop to the editor, as is shown in Figure A.18.

![Figure A.16. Opening diagram dialog](image)
Finally, to see the properties of the elements, such as the Id, Description, or Type, the user has to select the desired element; in the Properties tab, in the bottom of the window, the properties will be shown (see Figure A. 19).
A tool-supported method for fallacies detection in process-based argumentation

Figure A. 19. Show element properties