

Modeling Safety and Airworthiness (RTCA DO-178B) Information – Conceptual Model and UML Profile

Gregory Zoughbi ², Lionel Briand ¹, Yvan Labiche ²

¹ Simula Research Laboratory, P.O.Box 134, 1325 Lysaker, Norway

² Carleton University, 1125 Colonel By Drive, Ottawa, ON K1S5B6, Canada
greg@zoughbi.com, briand@simula.no, labiche@sce.carleton.ca

Abstract. Many safety-related certification standards exist for developing safety-critical systems. System safety assessments are common practice and system certification according to a standard requires submitting relevant system safety information to appropriate authorities. The RTCA DO-178B standard is a software quality assurance, safety-related, standard for the development of software aspects of aerospace systems. This research introduces an approach to improve communication and collaboration among safety engineers and software engineers in the context of RTCA DO-178B by proposing a Unified Modeling Language (UML) profile that allows software engineers to model safety-related concepts and properties in UML, the de-facto software modeling standard. A conceptual meta-model is defined based on RTCA DO-178B, and then a corresponding UML profile, which we call SafeUML, is defined to enable its precise modeling. We show how the SafeUML profile improves the line of communication between safety engineers and software engineers, for instance by allowing the automated generation of certification-related information from UML models. This is illustrated through a case study on developing an aircraft's navigation controller subsystem.

Keywords: UML, UML Profile, Conceptual Model, Meta-Model, Airworthiness, RTCA DO-178B, Safety, Safety-Critical, Safety Assessment, Certification

1. Introduction

Safety-critical systems must exhibit safe behaviour that does not contribute to hazards within the environment and context in which they are used. For example, an aircraft must allow the pilot to lift up the landing gears only if it is airborne. If the landing gears were lifted while the aircraft is on ground, then there would be a hazard, which is likely to result in damaging the aircraft and hurting its occupants. A hazard is a state of the system that could ultimately lead to an accident that may result in an injury or loss in human life.

Many standards require that a safety assessment be performed when developing or modifying a safety-critical system. Safety assessments, which have some similarities with risk assessments [11]

and are performed using similar methods, produce a list of safety requirements and constraints that the system (including its software part) must fulfil. RTCA DO-178B [22] is the de-facto safety-related standard for developing software in civil and military airborne systems. It provides guidance on how to achieve assurance levels that the software will not impact the continued safe flight of the aircraft.

Due to its impact on safety, and the need for safety assessments and thorough analysis, developing software for safety-critical systems is more expensive than developing software for non-safety-critical systems (some claim that to be in the order of 20 to 30 times more expensive [14]). One of the challenges, besides actually designing and implementing safety requirements allocated to software, is to be able to accurately communicate safety aspects among the different stakeholders involved in software development. This has already been documented in [5], where the authors reported on an extensive survey focused on DO-178B. This was performed by the NASA Langley Research Center (in collaboration with the Federal Aviation Administration (FAA)), to identify the challenges in developing software for safety-critical airborne systems. The authors claimed that correctly communicating requirements between different groups of people is the cause for many issues and difficulties encountered during the certification process of software intensive systems. They classified these difficulties in two major categories: (a) communication difficulties between regulatory people (e.g. certification authorities) and systems people (e.g. systems engineers and safety engineers), and (b) communication difficulties between systems people and software people (e.g. software engineers). The authors also found out that defining requirements is not an easy task – requirements can be complex and are usually inter-dependent. This certainly makes it even more difficult for different groups of people to communicate requirements accurately and effectively. For example, safety engineers, who are rarely software engineers, may define safety requirements that software engineers find infeasible or expensive to implement. If software engineers better understand the needs behind the requirements, then they may be able to propose solutions that are more practical and cost effective. On the other hand, software engineers may misinterpret the requirements due to their lack of experience in safety engineering and the application domain.

In this paper, we report on an approach to address these communication challenges in the context of DO-178B [22] while providing an open framework for tailoring to other industries, standards, organizations, and projects. The Unified Modeling Language (UML) [19, 20], is now the de-facto standard for modeling (object-oriented) software [21], and is increasingly used in the aerospace industry. Therefore, we aim at extending the UML notation with concepts inspired from DO-178B by defining a new profile that we call SafeUML. Because we focus on software for airborne systems in this research, the DO-178B standard [22] is analyzed to extract a list of key concepts that are of interest to both safety engineers and software engineers. We show that if those concepts are properly represented in UML models of software, then software engineers can document safety-relevant and certification-relevant information and decisions. A tool can then automatically generate reports

containing safety and certification information for DO-178B [22]. This provides safety engineers with better insights into the software compliance with safety and certification requirements, which they can easily track during the software development lifecycle. Those reports could also be used for safety cases or evidence of software compliance with the safety requirements, which can then be presented to external certification authorities. Furthermore, such a UML profile is expected to increase the software engineers' awareness of safety-related issues, which should enable them to better develop software for safety-critical systems, have a higher confidence in the developed software, and better communicate with safety engineers. Following a process similar to the two-staged approach proposed in [10], we define a conceptual model for our UML profile, explaining each concept, its attributes, and its relationships with other concepts. This approach mirrors the one used by the Object Management Group (OMG) to define its main profiles: Quality of Service (QoS) for high-quality and Fault-Tolerant (FT) systems [17], Schedulability, Performance, and Time (SPT) [18], and Modeling and Analysis of Real-Time Embedded (MARTE) systems [15]. Based on the conceptual model requirements, we design a UML profile, with stereotypes and tagged values, to model safety and certification information in the context of DO-178B [22]. We focus on class diagrams because they are the most commonly used diagram type, but future work will tailor the proposed stereotypes and tagged values to other diagrams such as sequence diagrams and state machines.

In an earlier paper [23], we explained the need for a UML profile for software development in the context of DO-178B [22], and then presented design examples that were modeled using our SafeUML profile. We now introduce the conceptual model of this UML profile in detail, explaining and justifying all the concepts, and then define the attributes of each concept as well as the relationships between the concepts. According to Lagarde *et al.* [10], the conceptual model is a key element in the design of UML profiles. Several UML profile designs can satisfy any given conceptual model, and we provide one carefully thought out design in this paper. In addition, we show how the SafeUML profile can help the development of software for safety-critical systems using DO-178B guidelines. For example, we discuss how certification information can be generated from models based on SafeUML to support various activities of software and safety engineers, and certification authorities.

The remainder of the paper is structured as follows. We discuss system safety and certification per DO-178B in section 2. In section 3, we present the requirements that a UML profile should have to adequately facilitate the definition and use of safety information in software models within the context of DO-178B, and then we identify and evaluate existing UML-based techniques for safety-critical systems in light of these requirements. We then present and explain our conceptual model in section 4, and define the SafeUML profile in section 5, while continuously validating the results with respect to development and certification per the DO-178B standard. A case study in section 6 presents a number of illustrating examples that show how SafeUML helps address DO-178B certification and communication challenges reported in [5]. Conclusions are drawn in section 7.

2. System Safety and Certification per DO-178B

We first provide a short introduction to DO-178B (section 2.1). We then present, based on the communication challenges reported in [5] and our own experience, how safety information is communicated between different stakeholders involved in the development of software-intensive systems (section 2.2). Then, we argue that embedding safety information into software engineering models facilitates the exchange of safety information between safety and software engineers (section 2.3), and we justify choosing UML models to embed safety information (section 2.4).

2.1. Introduction to DO-178B

Many industrial standards exist to help develop safety-critical systems. Some are common to all industry sectors (e.g., IEC 61508-3 [8]) whereas others are industry specific (e.g., CENELEC 50128 for Railway applications [2]). Herrmann summarizes them [6], and we discuss in [24] those that relate directly or indirectly to safety. One of those standards, RTCA DO-178B [22], is the de-facto safety-related standard for developing software aspects of aerospace systems. It is a quality standard outlining requirements on many aspects of software development for airborne systems.

DO-178B's requirements focus on the impact of software failure on safety, i.e. how critical is the failure of each component and what does that mean for the system safety as a whole. This impact is classified into five categories, namely: *Catastrophic*, *Hazardous/Severe-Major*, *Major*, *Minor*, and *No Effect*. Different certification or assurance levels exist for each failure category, namely level A, level B, level C, level D, and level E, respectively. Level A certification is the most rigorous and requires the submission of the largest amount of documents and proof of compliance with DO-178B. Level B is less rigid, followed by level C and then level D. Level E indicates that there is *No Effect* on flight safety and DO-178B requirements do not apply to it. As a result, the higher the certification level pursued, the more expensive it is to develop software in order to ensure higher confidence in the software quality and system safety.

The DO-178B standard defines failure condition categories and software levels based on the “severity of failure conditions on the aircraft and its occupants” [22]. This focuses on the aircraft's ability to undertake safe flights and excludes the safety impact on the environment outside the aircraft. The term “airworthiness” is therefore used, and indicates whether an aircraft is worth of a safe flight. This is more specific than Leveson's definition of safety, which was stated as “the freedom from accidents or losses” [11] without restriction to where the accidents or losses occur – inside or outside the aircraft. Therefore, there exists a difference between the concepts of “airworthiness” and “safety”. Airworthiness is concerned with accidents or losses on the *aircraft and its occupants* and is therefore a subset of safety: a safe system is airworthy but an airworthy system is not necessarily safe.

DO-178B requirements on software aspects of system certification include:

- Objectives of software lifecycle processes;

- Design and activity considerations to achieve those process objectives;
- Evidence consideration to prove that those process objectives are satisfied.

Examples of the DO-178B objectives and considerations include:

- Traceability of requirements for instance through design elements, implementation code, and object code (e.g. executable file generated by compiler). The depth of the traceability depends on the software's required assurance level;
- Software design and implementation approaches such as the use of software partitions, software replication, formal methods, configurable / loadable software, and the absence of recursive software.

2.2. Safety Information Usage Scenarios and Stakeholders

Studies were performed to improve the processes to develop safety-critical systems. One study of interest [5], conducted by National Aeronautics and Space Administration (NASA), investigated the challenges in software aspects of aerospace systems development using DO-178B. The authors reported communication challenges, which were consistent with our own separate experience.

To understand and present the challenges, we analysed what activities are performed by different stakeholders involved in the software development of safety-critical systems. We found that those activities concerned the storage, retrieval or analysis of safety-relevant information, which is continuously communicated across different stakeholder groups with different experience and skill sets. The use case diagram in Fig. 1 summarises these results, which we describe below:

Usage 1 – *Provide Safety Requirements*: Safety engineers perform a safety assessment of the system being developed. This results in safety requirements, a subset of which is allocated to software and communicated to software engineers.

Usage 2 – *Design Safety Requirements in Software*: Software engineers design the software system according to the safety requirements allocated to software.

Usage 3 – *Record and Justify Design Decisions*: Software engineers record and justify their design decisions. Traditionally, architectural and major design decisions are recorded in documents separate from the software model and detailed design decisions often appear as plain text comments in the source code, if at all. In practice, this makes it hard to ensure this information is systematically and precisely collected and documented and to automatically retrieve justifications for design decisions.

Usage 4 – *Monitor Safety*: Safety engineers continuously monitor the software system development activities with regards to the safety requirements they provided (Usage 1) during the software development lifecycle. To do so, they need to investigate how software engineers design the software (Usage 2), and why it is effective (Usage 3) to address the safety requirements they were provided with (Usage 1). Safety engineers can then analyze this information and discuss any issues with

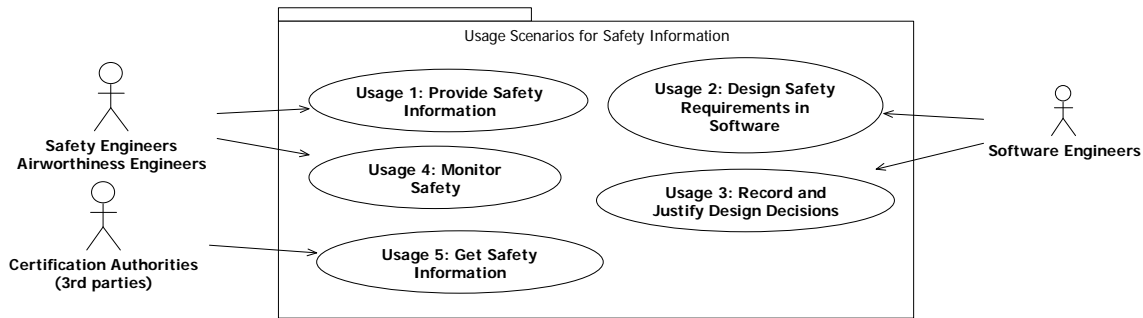


Fig. 1: Usage scenarios of safety information

software engineers, thus ensuring that system safety is continuously improving during the development lifecycle so that it meets its final safety objectives.

Usage 5 – *Get Safety Information*: Safety-relevant information, such as DO-178B-relevant information, is submitted to the appropriate authorities for certification, which usually occurs towards the end of the system development lifecycle. This information includes the safety requirements (Usage 1), the software design (Usage 2), the justification of the software design (Usage 3) given the safety requirements of the software (Usage 1), and the process used to continuously monitor the system safety during the development lifecycle (Usage 4).

2.3. A Case for a Model-Based Communication between Engineers

Studies have been performed on the current practices for communicating information among stakeholders. It was found out that this is mostly document-based, and such documentation tends to become large, ambiguous, inconsistent, and often lack clear structure [12]. The authors of [12] discuss the benefits from modeling information for safety-case development. In our work, the basic assumption is that our Usage scenarios (section 2.2) would be greatly facilitated if relevant safety information were captured and integrated in design models already in use for development. In other words, such augmented design model could play a central role in the precise communication of safety-related information across engineering groups and stakeholders. Software engineers could thus record such information in models and safety engineers could monitor it through reports automatically generated from models using appropriate tools. For reasons further explained, our work will express these models in an extension (profile) of the Unified Modeling Language (UML) that we will specifically define to address the needs of DO-178B. However, safety engineers would not need to understand the specifics of UML because any tool that extracts information from the model can present it in a model-independent format. To certify the system, a tool can automatically generate safety and certification information from UML models. This information would be produced in a format suitable for submission to the certification authorities (e.g. DO-178B specific terminology).

While this will become clearer when going through our case study, here is a summary of the practical benefits we expect for each usage scenario in Fig. 1:

- Safety engineers will better communicate requirements to software engineers (Usage 1).
- Software engineers will be able to better indicate how their design decisions relate to safety and certification aspects (Usage 2 and Usage 3).
- Safety engineers will be able to monitor system safety throughout the system development by easily and quickly generating reports from UML models. Such reports would explain how and why software engineers designed the safety requirements in the system (Usage 4).
- Safety engineers will easily and quickly generate certification-required information at any point during the system development lifecycle (Usage 5). This becomes an automated process and reduces efforts of safety engineers to collect safety information. As a result, they are now more available to focus on safety-added activities, as opposed to administrative ones, such as safety and certification requirements and strategies.

2.4. A Modeling Approach based on UML

Our approach to better support communication among stakeholders for safety design, analysis and certification activities is to define a specific UML profile named SafeUML [21]. We focus on the DO-178B standard [22] because it is a very popular standard for developing software for certifiable airborne systems. Therefore, SafeUML will model concepts and information relevant to DO-178B. However, we made a conscious effort to keep our approach as generic as possible to facilitate tailoring to different industries, organizations, and projects. Furthermore, airworthiness is a subset of safety and therefore many of the concepts involved should be reusable for other safety domains. Nevertheless, we consistently refer to DO-178B when we validate our approach throughout this paper.

For example, one of the certification requirements of DO-178B is the submission of a project's Plan for Software Aspects of Certification (PSAC). This plan should include information as described in section 11 of DO-178B [22]. When we validate our research results (conceptual model and UML profile), we often refer to the PSAC and specify how information required in that document can be generated from a UML model using our proposed SafeUML profile.

Languages other than UML could have been considered for this work. However, there are many benefits [24] justifying the use of UML, a common standard for modeling software, including:

1. Many development environments and methodologies support the use of UML.
2. Extension mechanisms are part of the language to help extend it while being able to use existing modelling environments.
3. UML is known in the aerospace industry, thus lowering the cost of training and adoption.
4. UML will help increase the precision of safety information due to the possibility, through extension mechanisms, of making the semantics of UML constructs more precise and specific to a domain or an industry.

5. UML is part of the OMG recommended, Model-Driven Architecture (MDA), which is a popular approach for the development of airborne systems (e.g., [1]).

3. Assessment of Existing UML-Based Solutions

In this section, existing UML-based solutions that allow the modelling of safety aspects in UML are reviewed (section 3.2) based on a set of requirements that we derived by analysing the DO-178B standard (section 3.1).

3.1. Safety-Related Information Requirements

We performed a careful analysis of the DO-178B [22] standard, which led us to identify relevant DO-178B-related concepts. Such concepts should somehow be captured by a suitable UML-based solution to be used. Since airworthiness is a subset of safety (see section 2), all of the airworthiness-related concepts we identified are also safety-related concepts and are referred to as such in the remainder of this document. Though not restricted to airworthiness, those concepts are however clearly not the only ones needed for other safety-critical industries including transportation, medical, and nuclear.

A careful analysis of the DO-178B [22] standard led us to identify safety-related concepts that were found relevant to consider when developing software for airborne systems. Such concepts are not all directly related to safety, hence the use of the term “safety-related concepts” rather than simply “safety concepts”. Examples include concepts related to reliability (e.g. fault-tolerance), performance, and certification. Those safety-related concepts were refined into a smaller set of safety-related concepts that capture the same information in a more concise manner. Though this is discussed in detail in section 4.1, it should be noted that those refined concepts became the basis for the information requirements that a suitable UML-based solution must satisfy. The detailed list of the safety-related concepts can be found in [24], although the conceptual model presented in section 4.2 shows the refined safety-related concepts.

Based on the safety-related concepts, we carefully produced 54 information requirements that identified the information that a suitable solution must be able to model in UML. Therefore, they became the basis upon which solutions to model safety-related concepts in UML were assessed. Complete information requirements are found in [24] and examples are provided below:

- IREQ 1 The solution shall be able to identify a safety-related context under which safety information, specified through other means, is valid
- IREQ 9 The solution shall be able to specify software requirements, including safety-related, and certification-related
- IREQ 11 The solution shall be able to model a software model deviation from a plan, requirement, or a standard

- IREQ 12 The solution shall be able to model specific software implementation styles of interest to DO-178B-related software such as recursion, dynamic memory, compacted expressions, and data aliases
- IREQ 15 The solution shall be able to model COTS software, including the rationale for using it
- IREQ 20 The solution shall be able to model software events
- IREQ 21 The solution shall be able to specify how a particular event affects system safety
- IREQ 22 The solution shall be able to model software reactions, or responses, to software events.
- IREQ 25 The solution shall be able to model safety-critical elements
- IREQ 26 The solution shall be able to specify the criticality level of safety-critical model elements, or the element's contributions to failure conditions
- IREQ 27 The solution shall be able to model a software partition
- IREQ 30 The solution shall be able to model safety monitoring software
- IREQ 39 The solution shall be able to model hardware/software interfaces
- IREQ 52 The solution shall be able to model multiple-version dissimilar software
- IREQ 53 The solution shall be able to model software comparators, or voters, for multiple version dissimilar software
- IREQ 54 The solution shall be able to specify the voting policy parameters for software comparators, or voters.

3.2. Existing UML-based Solutions

There exist several UML-based solutions that can be considered to capture safety-related information in UML models.

The Object Management Group (OMG) released a profile to model Quality of Service (QoS) for high-quality and Fault-Tolerant (FT) systems [17]. It includes frameworks to describe quality of service, risk assessment, and fault tolerance. The quality of service part can be used to model safety. The risk assessment framework provides mechanisms for modeling risk contexts, stakeholders, weaknesses, opportunities and threats, unwanted incidents, risk quantification, risk mitigation and treatments. The fault tolerance framework includes mechanisms for describing fault-tolerant software architectures in general as a technical solution to reliability requirements, and focuses on modeling software redundancy, or software replication.

The Schedulability, Performance, and Time (SPT) profile [18], which provides mechanisms to model concepts of importance to real-time systems was also released by the OMG. It includes frameworks to model resources, time, concurrency, schedulability, and performance, and allows developers to perform model-based performance analysis. It does not focus primarily on safety. More recently, OMG has released a Beta version for a Modeling and Analysis of Real-Time Embedded

(MARTE) systems profile [15], which was still being finalized pending a final release at the time of writing this paper (latest version is MARTE 1.0 Beta 2). The MARTE profile is planned to supersede the SPT profile once it is final, but SPT was the official UML profile for schedulability, performance, and time when this research was performed and hence we used the SPT profile instead.

The High Integrity Distributed Object-Oriented Real-Time Systems (HIDOORS) was a European project [13]. One of its goals was to introduce SPT profile-compliant mechanisms for modeling safety-critical and embedded real-time applications. Although aimed at safety-critical applications, it specialised SPT concepts such as triggers, actions, resources, and scheduling jobs by focusing on concurrency, performance, and schedulability. It did not specifically focus on safety – the freedom from accidents and losses.

Jan Jürjens presented a UML profile [9] that aimed at modeling reliability aspects regarding transmitting messages (e.g., maximum failure rates for message communication). Jürjens argued that since failures related to lost, delayed, or corrupted messages have an impact on safety in safety-critical applications, the profile can be used for developing safety-critical applications. It included mechanisms to model risks, crashes, guarantees, redundancy, safe links, safe dependencies, safety critical elements, safe behaviours, containment, and error handling.

Based on the IEC 61508 standard [8], Hansen and Gullesten presented a series of UML patterns that can be used to model some aspects of safety-critical systems [4]. The patterns allowed modeling the safety quality of service, software diversity and voting, partial diversity with built-in diagnostic or monitoring, “safe” communication protocols, and some other topics such as testing, hazard analysis and quality development. They include mechanisms to model, in use cases, redundancy, monitoring, and voting based on multiple output comparisons.

We evaluated how the UML-based solutions discussed above score with respect to satisfying the information requirements discussed in section 3.1 – the full list of information requirements is available in [24]. Each profile’s score was calculated based on how many information requirements it met. We observed (see details in [24]) that none of the existing solutions that we evaluated achieved more than 31% of the maximum score. In fact, all of the solutions combined only met 44% of the information requirements. For instance: events and reactions can be modeled in HIDOORS [13] and SPT [18], but not in the other three solutions; monitors can only be modeled in [9] and [18]; a model element’s contribution to failure conditions (i.e., criticality levels in the DO-178B standard) is only handled in [4], [9], and [18]; the implementation style (e.g., recursion, dynamic memory allocation), hardware/software interfaces, and configurable components cannot be modeled by any of the existing solutions, and those are all information requirements identifies in [24]. We therefore decided to develop our own SafeUML solution for airworthiness, as a UML profile, as described next.

4. Safety-Related Conceptual Model for Airborne Systems

In this section, we detail the conceptual model used to describe software considerations for airborne systems. We first describe the process we followed (section 4.1) and then the conceptual model we defined (section 4.2). The conceptual model is a general data model for information needed to develop software in compliance with the DO-178B standard [22], and it can be reused in approaches other than a UML profile (e.g. DOORS or Oracle databases).

4.1. Design Process

Our conceptual model design approach is consistent with the approach devised by Lagarde *et al.* [10]. Consistent with step 1 of their process, we reviewed the DO-178B standard [22] and identified the raw list of concepts (see section 3.1 in [24] for further details). Consistent with step 2 of their process, we refined the concepts by reducing the problem space – we removed irrelevant concepts and redundancies, as detailed in [24]. Consistent with step 4 of their process, we also went through an iterative optimization process to:

1. Remove duplicate concepts: Seemingly different concepts sometimes appear in the DO-178B standard where, in reality, they represent the same idea. For instance, *Multiple-Version Dissimilar Software* and *Software Redundancy* describe the same idea of using multiple software components that have the same functionality but different implementations.
2. Group concepts: Some concepts are in fact examples of a more general concept. For instance, *Safety Monitoring*, *Fault Detection*, *Integrity Check* are applications of a single software-concept that is “Monitor”, which monitors the activity of other components to detect unusual, potentially hazardous, events or behaviours. Thus, they are grouped as a single concept, and they are differentiated via attributes (i.e., a “Monitor” concept with a “Kind” attribute that is “Safety” for *Safety Monitoring*, “Reliability” for *Fault Detection*, and “Integrity” for *Integrity Check*).
3. Precisely define concepts: Each concept is clearly defined, with attributes each describing a single aspect of the concept. For instance, when defining the “Requirement” concept, we give it an attribute called “Specification” that can be used to specify what the requirement is (e.g. “Speed > 20 m/s”). We use a consistent approach for concept definition, adapting approaches used in existing UML profiles (e.g., [15], [17], and [18]). For each concept, we provide: concept definition, attribute definition, relations to other concepts, and reference to the original DO-178B concept(s). This paper provides a concise version of the concept definition without omitting any information (see section 4.2 and Appendix A), but more detailed definitions are used in [24].

This activity resulted in the definition of 26 safety-related concepts, 48 attributes, and 35 inter-concept relationships. Their various combinations can represent all of the original safety-related concepts obtained from DO-178B, and satisfy all information requirements. These refined concepts

and their relationships were defined and then grouped according to their purpose, and finally formalized under the form of a conceptual model represented as a UML class diagram.

4.2. Conceptual Model

The refined concepts were grouped into five different categories depending on their purpose. These categories were determined after an analysis of all final concepts obtained as described in section 4.1:

1. *Requirements*: This category contains concepts to model requirements, deviation from requirements, implementation styles, behavioural styles, source of the design (e.g. COTS, previously developed, ... etc), requirements traceability through design and code, and partitioning decisions.
2. *Characteristics*: This category contains concepts to designate safety-critical elements and identify their software levels or failure condition categories, simulators to model elements, design strategies (e.g. round-robin, FIFO, ... etc), measures such as complexity and coupling, and hardware/software interfaces.
3. *Event Management*: This category contains concepts to model events and actions that may impact system safety, responses to such events or actions, where they are detected, and where they are handled to ensure safety. It also includes concepts to designate inputs defended against using defensive programming and identifies active and passive elements.
4. *Configuration*: This category contains concepts to model software configurations and user-modifiable software, including which design elements are configurable, which ones can be loaded on platforms to change configurations, and where in the design such changes are initiated and controlled.
5. *Replication*: This category contains concepts to model software redundancy, e.g. using multiple-version dissimilar software, and allowing the modeler to designate redundant elements and voting elements.

Each concept, along with its attributes and its relationships to other concepts, are discussed in detail in Appendix A. However, we concisely introduce each concept in Table 1: they are all numbered C1 to C26. The conceptual model, i.e. the concepts, their relationships and the five groups discussed above, is represented as a class diagram (with components for the groups), as illustrated in Fig. 2.

Table 1: Concepts of our conceptual model (short descriptions)

Category	Concept
Requirements	Requirement (C.1): specifies a requirement, safety or not, that must be met, and it may be derived and/or traceable to a higher-level requirement
	Deviation (C.2): identifies a design deviation from a plan, standard, or Requirement (C.1)
	Style (C.3): an abstract concept indicating an implementation or a behavioural style
	Implementation Style (C.4): identifies a style that is used to implement a design
	Behavioural Style (C.5): identifies and describes a behavioural style of a design

Category	Concept
	Nature (C.6): describes the source for the design (e.g. COTS, previously developed)
	Rationale (C.7): describes the reason of existence (e.g. requirement) of a particular design
	Partition (C.8): identifies a design partitioned from another for safety purposes
Characteristics	Safety Critical (C.9): a design element that impacts system safety
	Environmental Model (C.10): models/simulates behaviour of Safety-Critical (C.9) element
	Strategy (C.11): describes strategy (e.g. safety, reliability, scheduling) used in design
	Measure (C.12): quantifies a characteristic of a design element, including complexity
	Interface (C.13): describes an interface, e.g. hardware/software interface
Event Management	Event (C.14): describes safety-relevant event or action that may occur
	Reaction (C.15): describes a response to one or more Events (C.14)
	Handler (C.16): receives Events (C.14) and performs relevant Reactions (C.15)
	Monitor (C.17): monitors Safety-Critical (C.9) elements, detects Events (C.14), and notifies Concurrent (C.18): specifies active/passive modes with respect to triggering Events (C.14)
	Defensive (C.19): identifies inputs defended against and corresponding Reactions (C.15)
Configuration	Configuration (C.20): represents a software/hardware configuration (e.g. lookup tables)
	Configurable (C.21): identifies an element that is altered to produce a Configuration (C.21)
	Loadable (C.22): identifies an element that can be loaded to produce a Configuration (C.21)
	Configurator (C.23): identifies an element that changes Configurations (C.21)
Replication	Replicated (C.24): identifies an element that participates in a Replication Group (C.26)
	Comparator (C.25): compares outputs of Replicated (C.24) elements in a Replication Group (C.26) and determines a suitable output (e.g. based on a majority voting scheme)
	Replication Group (C.26): identifies a group of redundant/Replicated (C.24) elements for reliability (e.g. voting-based redundancy group of similar elements developed separately)

As it is evident from the concepts introduced above, including the suggested examples for the attribute values in Appendix A, the conceptual model is not specific to airworthiness, which suggests that it and the profile we describe later in this paper could be reused and extended in other safety-

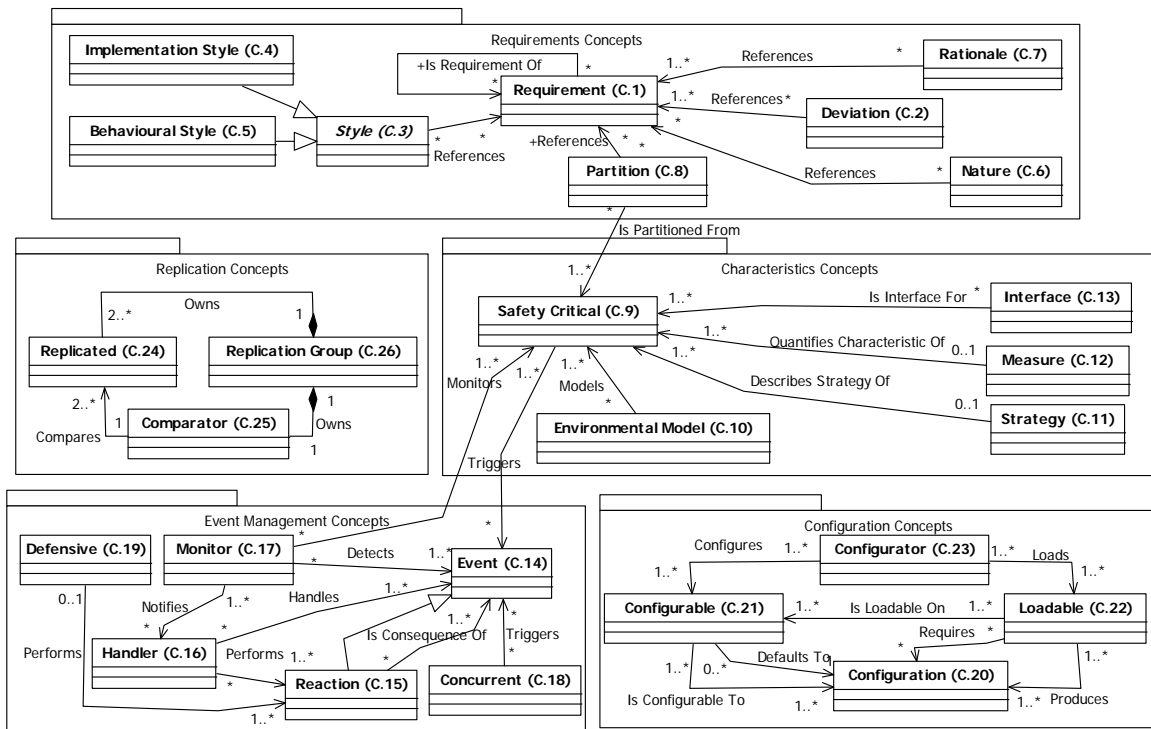


Fig. 2: Conceptual model for safety-related concepts

critical industries including health care, transportation, and nuclear. For example, the “SafetyCritical” concept has a “criticality level” attribute whose definition can refer to the DO-178B standard as well as the IEC 61508 standard [8]. This illustrates our conscious effort to be as general as possible while maintaining DO-178B requirements as the priority. Additionally, the attributes’ values are not fixed and can be organization (or project) specific, e.g., the “High”, “Medium” and “Low” values for attribute “Confidence Level” can be assigned industry specific meanings. This will translate into a flexible UML profile, with customizable tagged values.

Table 2 identifies examples of the original DO-178B [22] concepts, the DO-178B section numbers in which they appear, and where such information is captured in our conceptual model. This validates that our conceptual model is in fact designed to model concepts relevant to DO-178B [22].

Table 2: Concepts are tied to DO-178B (samples)

Concept	DO-178B Concept (and Section Number)
Requirement (C.1)	Safety Objective (4.1), Safety Requirement (2.1.1, 5.1), Certification Requirement (2.1.1), Derived Requirement (5.1.1), Design by Contract
Deviation (C.2)	Deviation (8.2), Accuracy (11.9)
Style (C.3)	Implementation Style, Time-Related (6.4.2.1), State-Related (6.4.2.1)
Implementation Style (C.4)	Recursion (6.3.3, 11.7), Compacted Expression (11.7), Dynamic Memory (11.7), Data Alias (11.7), Discontinuity (6.3.2)
Behavioural Style (C.5)	Time-Related (6.4.2.1), State-Related (6.4.2.1)
Nature (C.6)	COTS Software (2.4, 11.1), Deactivated Code (4.2, 5.4.3, 11.10), Previously Developed Software (11.1, 11.3, 12.1)
Rationale (C.7)	Traceability (5.3.1, 5.5)
Partition (C.8)	Partitioning (2.1.1, 2.3.1, 5.2.2, 6.3.3, 11.3, 11.9, 11.10)
Safety Critical (C.9)	Safety-Critical, Software Level (2.2, 2.2.2), Level of Confidence (4.1, 6.4), Failure Condition Category (2.2, 2.2.1)
Environmental Model (C.10)	Simulator (12.3.3.5)
Strategy (C.11)	Safety Strategy (2.1.1), Scheduling Strategy (11.1), Formal Method (12.3.1)
Measure (C.12)	Complexity (5.2.2, 6.3.4, 11.7), Coupling (11.8)
Interface (C.13)	Hardware / Software Interface (6.4.3, 11.1, 11.9)
Event (C.14)	Unsafe Action, Failure (2.2), Failure Condition (2.2), Fault (2.1.1), Error (4.2), Integrity Check (11.16)
Reaction (C.15)	Safety Response (2.1.1)
Handler (C.16)	Exception Handling (11.7), Fault Containment (2.1.2), Software Protector (2.4), Safety Feature, Fault Tolerance (2.1.1, 4.4, 11.1), Immunity (2.1.1)
Monitor (C.17)	Safety Monitoring (2.1.1, 2.3.2, 11.9), Error Detection (4.2), Fault Detection (2.1.1, 11.9), Fault Containment (2.1.2), Error Prevention (4.2, 4.4), Integrity Check (11.16), Software Protector (2.4), Loadable Software Indicator (2.5), Safeguard (7.2.8, 11.4), Safety Feature (4.4, 11.1)
Concurrent (C.18)	Active (12.3.3), Passive (11.7, 12.3.3), Shared Resource (11.1), Multi-Tasking (11.7)
Defensive (C.19)	Defensive Programming (4.5)
Configuration (C.20)	Configuration (5.4.3)
Configurable (C.21)	User Modifiable Software (2.4, 4.2, 5.2.3, 11.1), Option Selectable Software (2.4, 11.1)
Loadable (C.22)	Field Loadable Software (2.5, 6.4.3, 11.1), Software Patch (5.4.3)
Configurator (C.23)	Loader (2.5), Loadable Software Indicator (2.5)
Replicated (C.24)	Multiple-Version Dissimilar Software (2.1.1, 2.3.2, 11.1, 11.3), Software Redundancy (2.1.1, 11.1)
Comparator (C.25)	Comparator/Voter (2.3.2)

Concept	DO-178B Concept (and Section Number)
Replication Group (C.26)	Multiple-Version Dissimilar Software (2.1.1, 2.3.2, 11.1, 11.3), Software Redundancy (2.1.1, 11.1)

5. SafeUML – Safety-Related UML Profile for Airborne Systems

This section introduces SafeUML, a UML profile that satisfies the information requirements discussed in section 3.1, and which is based on the conceptual model described in section 4. Since we did not find an existing UML solution that met more than one third of the information requirements, we proceed by designing one, specifically a UML profile, to satisfy all of them. As described in [10], SafeUML may be seen as one possible implementation of the defined conceptual model.

The profile name, SafeUML, suggests that the UML profile targets software for safety-critical systems. Though this profile was developed to help certification under DO-178B requirements, SafeUML is designed to be as general as possible and tailorable to different standards, organizations, and projects. In short, it is a generic profile whose primary and original motive is DO-178B.

5.1. Profile Design

Our profile, which implements the conceptual model presented in section 4.2, resulted in 31 stereotypes with 79 tagged values. We explained in section 1 that we first focused on class diagrams because they are the most commonly used diagram type. Stereotypes and their associated tagged values will apply to several meta-model elements of the class diagram including classes, operations, and relationships, but also to packages as certain properties can apply to a group of elements (e.g., subsystem). Therefore, we apply each of our stereotypes on instances of the following meta-classes, specified in [19] and [20]:

1. Kernel::Package, or simply *Package*
2. Kernel::Class, or simply *Class*
3. Kernel::Operation, or simply *Operation*
4. Kernel::Relationship, or simply *Relationship*

Fig. 3 illustrates that our SafeUML profile package is composed of six sub-packages as describes below (Fig. 4 illustrates naming conventions used for visual simplicity):

1. *ContextStereotypes*: This package (Fig. 5(a)) contains stereotypes to indicate that packages or diagrams, generally containing several closely-related design elements, contain information relevant to a specific context, e.g. safety context or reliability context.
2. *RequirementsStereotypes*: This package (Fig. 6) contains stereotypes to model concepts specified in category Requirements (see section 4.2)
3. *CharacteristicsStereotypes*: This package (Fig. 8) contains stereotypes to model concepts specified in category Characteristics (see section 4.2)

4. `EventManagerStereotypes`: This package (Fig. 9) contains stereotypes to model concepts specified in category Event Management (see section 4.2)
5. `ConfigurationStereotypes`: This package (Fig. 7) contains stereotypes to model concepts specified in category Configuration (see section 4.2)
6. `ReplicationStereotypes`: This package (Fig. 5(b)) contains stereotypes to model concepts specified in category Replication (see section 4.2)

Table 3 precisely introduces each stereotype.

Table 3: Definition of stereotypes in SafeUML

Package	Stereotype
Context Stereotypes	<code><<SafetyContext>></code> (S.1): identifies a safety-related information context
	<code><<ReliabilityContext>></code> (S.2): identifies a reliability-related information context, e.g. it can be used to describe or identify a specific replication group (see related concept). Also, same as “Replication Group” (C.26) concept
	<code><<IntegrityContext>></code> (S.3): identifies a integrity-related information context
	<code><<PerformanceContext>></code> (S.4): identifies a performance-related information context
	<code><<ConcurrencyContext>></code> (S.5): identifies a concurrency-related information context
	<code><<CertificationContext>></code> (S.6): identifies a certification-related information context
	<code><<DesignContext>></code> (S.7): identifies a specific design-related information context
	<code><<ConfigurationContext>></code> (S.8): identifies a configuration-related information context, e.g. it can be used to describe or identify a specific configuration. Also, same as “Configuration” (C.20) concept
Requirements Stereotypes	<code><<Requirement>></code> (S.9): same as “Requirement” (C.1) concept
	<code><<Deviation>></code> (S.10): same as “Deviation” (C.2) concept
	<code><<ImplementationStyle>></code> (S.11): same as “Implementation Style” (C.4) concept
	<code><<BehaviouralStyle>></code> (S.12): same as “Behavioural Style” (C.5) concept
	<code><<Nature>></code> (S.13): same as “Nature” (C.6) concept
	<code><<Rationale>></code> (S.14): same as “Rationale” (C.7) concept
Characteristics Stereotypes	<code><<Partition>></code> (S.15): same as “Partition” (C.8) concept
	<code><<SafetyCritical>></code> (S.16): same as “Safety Critical” (C.9) concept
	<code><<EnvironmentModel>></code> (S.17): same as “EnvironmentModel” (C.10) concept
	<code><<Strategy>></code> (S.18): same as “Strategy” (C.11) concept
	<code><<Measure>></code> (S.19): same as “Measure” (C.12) concept
EventManager Stereotypes	<code><<Interface>></code> (S.20): same as “Interface” (C.13) concept
	<code><<Event>></code> (S.21): same as “Event” (C.14) concept
	<code><<Reaction>></code> (S.22): same as “Reaction” (C.15) concept
	<code><<Handler>></code> (S.23): same as “Handler” (C.16) concept
	<code><<Monitor>></code> (S.24): same as “Monitor” (C.17) concept
	<code><<Concurrent>></code> (S.25): same as “Concurrent” (C.18) concept
Configuration Stereotypes	<code><<Defensive>></code> (S.26): same as “Defensive” (C.19) concept
	<code><<Configurable>></code> (S.27): same as “Configurable” (C.21) concept
	<code><<Loadable>></code> (S.28): same as “Loadable” (C.22) concept
Replication Stereotypes	<code><<Configurator>></code> (S.29): same as “Configurator” (C.23) concept
	<code><<Replicated>></code> (S.30): same as “Replicated” (C.24) concept
	<code><<Comparator>></code> (S.31): same as “Comparator” (C.25) concept

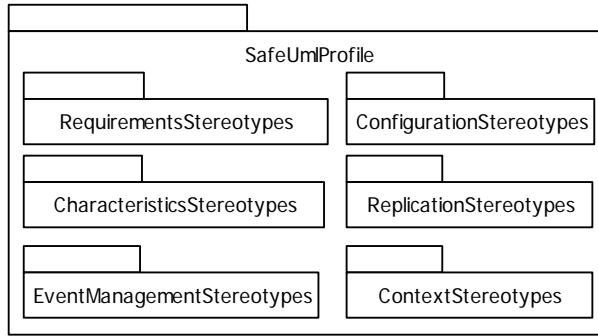


Fig. 3: Main packages of SafeUML Profile

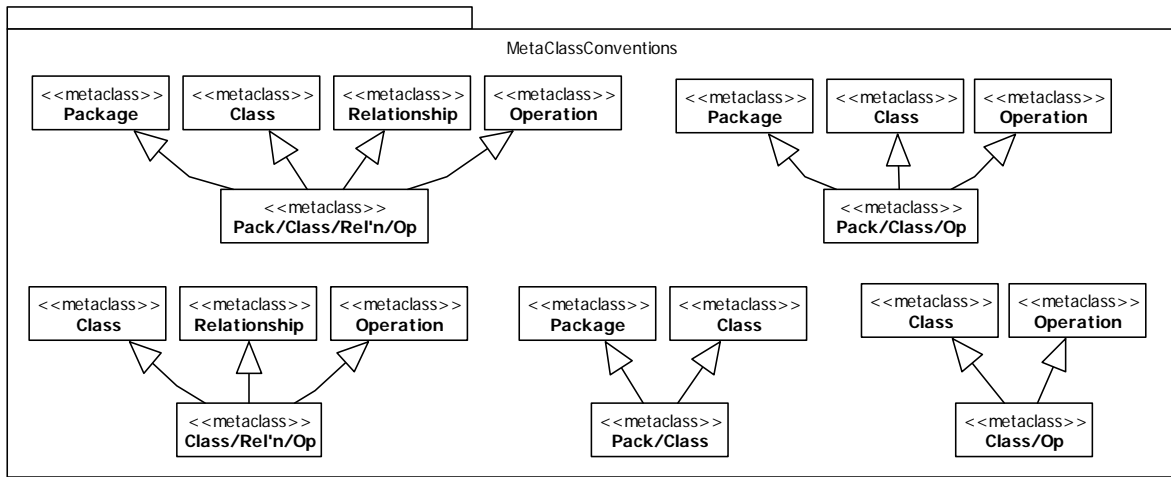


Fig. 4: Meta-Classes Used for Visual Simplicity

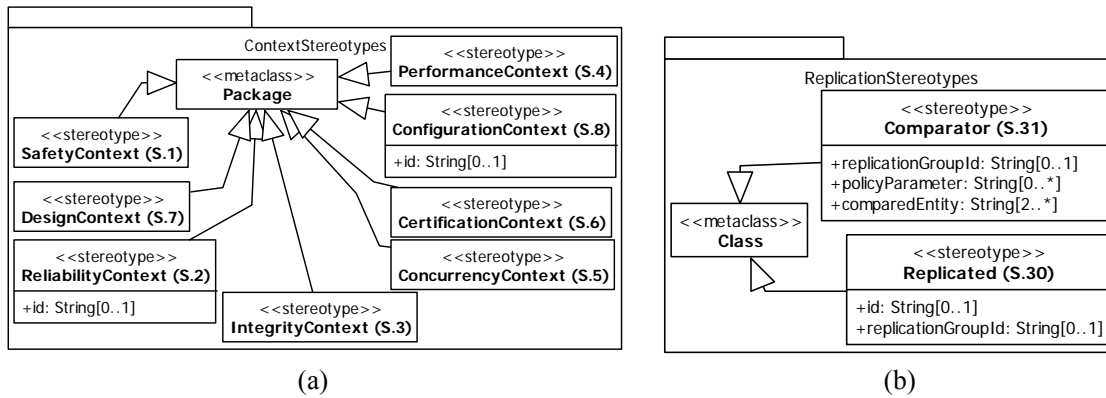


Fig. 5: Packages contextStereotypes (a) and ReplicationStereotypes (b)

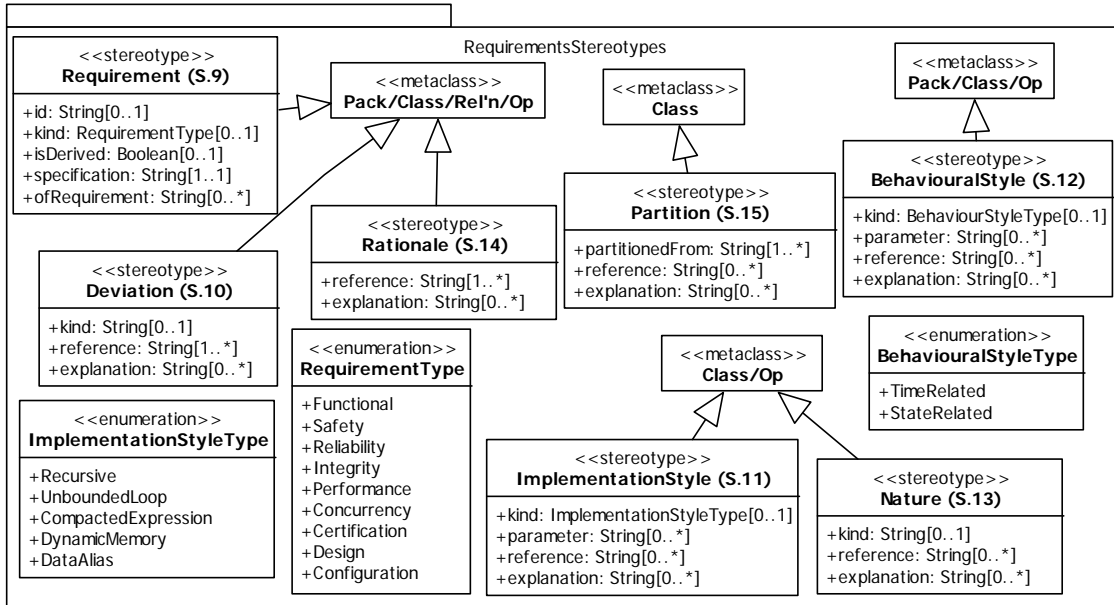


Fig. 6: Package RequirementsStereotypes

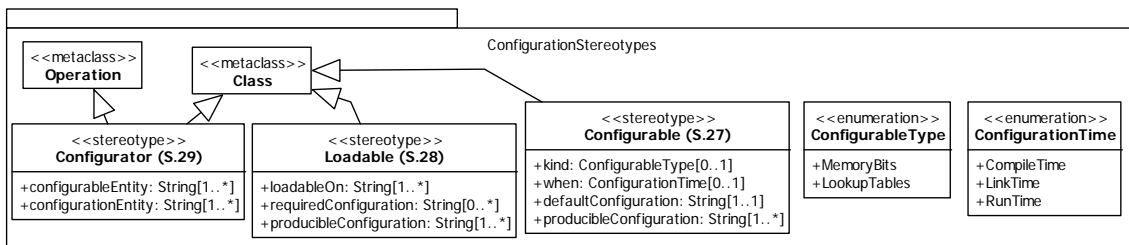


Fig. 7: Package ConfigurationStereotypes

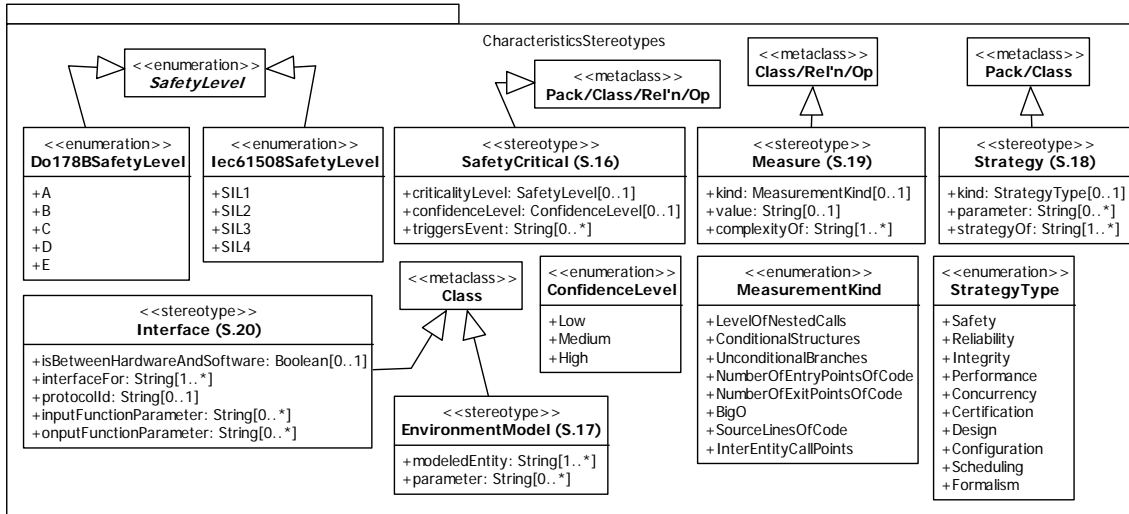


Fig. 8: Package CharacteristicsStereotypes

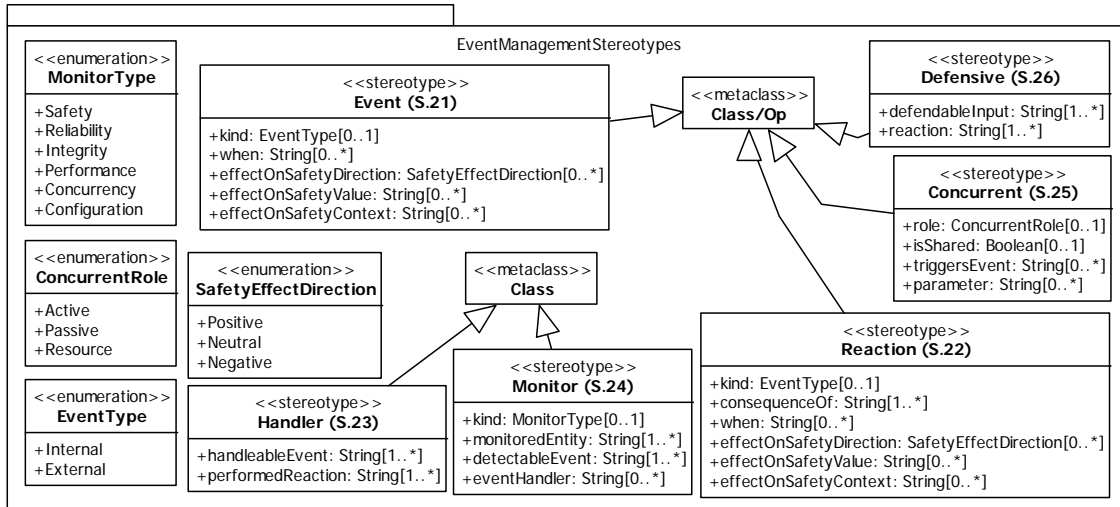


Fig. 9: Package EventManagementStereotypes

In the SafeUML profile, we model enumerations as classes. Sometimes, there are more than one set of enumerations that can be applied. For example the level of safety-critical model elements can depend on the applicable safety-related standard. The DO-178B standard [22] uses (A, B, C, D, E) whereas IEC 61508 [20] uses (SIL1, SIL2, SIL3, SIL4). That is why class `SafetyLevel` has two possible sub-classes, one for each standard. Additional enumerations can be added by introducing new subclasses for project-specific requirements.

Recall that our profile design is consistent with the approach devised by Lagarde *et al.* [10]. Per steps 3 and 4 of their process, we transformed the conceptual model elements into potential stereotypes. Each stereotype inherits from UML meta-classes to indicate which kinds of model elements it can be applied on (`Package`, `Class`, `Operation`, `Relationship`). Then, the resulting stereotypes were optimized to remove relationships that violated meta-relationships in the UML meta-model among the four base classes listed above (`Package`, `Class`, `Operation`, `Relationship`), and replace them with attributes. As a consequence, some concept relations (from Fig. 2), and therefore stereotypes relations, do not explicitly appear in the packages. For instance, a `Reaction` (C.15) reacts to an `Event` (C.14) (or another reaction) and there is a relationship between them. However, there is no association between stereotype `Reaction` (S.22) and stereotype `Event` (S.21) since there is no association between the meta-classes those two stereotypes inherit from in the UML 2.0 metamodel (Fig. 9). Instead, there is an attribute `consequenceOf` in stereotype `Reaction` (S.22) that lists the event names (or reaction names) that `Reaction` (S.22) reacts to. Hence, SafeUML remains consistent with the UML meta-model.

Using one of the existing Object Constraint Language (OCL) engines, for example Model Development Tools (MDT)-OCL, working on Ecore models in Eclipse Modeling Framework (EMF), from IBM [7], the model can be queried using OCL [16] expressions. For instance, a `Reaction` (S.22) is a `Class` or `Operation` that implements a reaction to an `Event` (S.21) (or another

Reaction (S.22)), and this is modelled using the consequenceOf attribute values in the SafeUML profile. This information can be retrieved by executing an OCL query, using an OCL engine, on an instance model of the SafeUML profile. In the context of Reaction (S.22), the query is defined as:

```
self.consequenceOf->forAll(s:String|
    Event.allInstances->including(Reaction.allInstances)->exists(name=s)
```

5.2. Value of Information Captured by SafeUML

This section discusses the value of information that can be generated from SafeUML models by relating it to the usage scenarios defined in section 2.2 for safety engineers, certification authorities, and software engineers. SafeUML is designed to enable modeling information relevant to DO-178B [22]. As Fig. 10 illustrates, DO-178B contains various concepts of importance. These concepts were identified and captured in a conceptual model (see section 4.2, which includes a mapping from the conceptual model to the original DO-178B concepts). In turn, the conceptual model was then implemented by the SafeUML profile (see section 5.1, which includes a mapping from each SafeUML stereotype to the corresponding concept from the conceptual model). It then follows that SafeUML captures all DO-178B-relevant information.

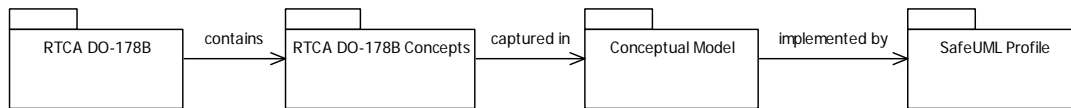


Fig. 10: SafeUML Relation to DO-178B

Usage 1 – Usage 3 illustrate that safety-related information is communicated and documented. Next, Usage 4 illustrates the need for continuous monitoring of safety and certification aspects during the development lifecycle. Critical to this is achieving effective communication between safety engineers and software engineers. Since SafeUML captures DO-178B-relevant information, SafeUML models can be used as a repository for these two groups to exchange precise, unambiguous information regarding software-based system safety. By simply performing their regular design modeling duties, software engineers will collect and formulate safety and certification information in their designs. Once captured by the modeling tool, this information can be easily generated at any point of time during the development lifecycle using simple tool support to query the models and generate customized reports. Safety engineers can thus obtain regular reports, monitor the progress of implementing safety requirements, and promptly address any concerns. This will put them in position to be better prepared to defend the system’s compliance to DO-178B. For example, they can use this approach to assess the availability of certification-related information.

Usage 5 in section 2.2 illustrates the need to obtain valuable information from the model that can be submitted to certification authorities. Since the SafeUML conceptual model was based on the DO-178B standard and its requirements, the information captured in SafeUML models should fully address the need to demonstrate compliance with the standard. For example, section 11 and annex A

in the DO-178B [22] standard list the information required for submission to the certification authorities for each software level. This was accounted for and included in the information requirements in section 3.1 and therefore captured in the conceptual model in section 4.

For example, the notions of requirements refinement and derived requirements are very important to DO-178B [22]. Safety requirements are assigned at a system functional level and then they are refined or decomposed into more detailed requirements, some of which are then assigned to software design elements. Such requirements refinement is an important aspect of requirements engineering. In addition, the system development team may introduce new requirements or derive them from the customer requirements and, because they are not explicitly specified by the customer, they are called derived requirements [22]. Such derived requirements often result from environmental conditions, standards, or development processes. SafeUML supports such decomposition and the modeling of derived requirements through its stereotypes and tagged values. SafeUML has a `<<Requirement>>` stereotype, which is used to identify a requirement. This stereotype has a tagged value called `ofRequirement`, which identifies its parent requirement (refinement relation). It also has a tagged value called `isDerived`, which identifies whether the requirement is a derived one or not. At any point in the requirements hierarchy, refined or derived requirements may be traced to higher-level requirements (through the `ofRequirement` relationship). For example, Fig. 11 shows three safety requirements for an aircraft’s payload release system and their relationships. SREQ 1 specifies that the software system shall safely release payloads from the aircraft. This requirement is refined in SREQ 1.1, which states that the payload released from the aircraft bay shall not hit the aircraft’s body while being released: SREQ 1.1 is linked to SREQ 1 by a `ofRequirement` link. Then, through analysis of the weight of payloads and aircraft, safety engineers use the physics laws of free fall and determine the maximum speed an aircraft may be flying such that the released payload does not hit the body. This leads to derived requirement SREQ 1.1.1: Payload shall not be released from the aircraft if its speed is greater than 300 knots; as illustrated by the `isDerived=true` attribute of SREQ 1.1.1. Notice that the derived requirement appears in the form of a constraint (on the speed), which is common in practice.

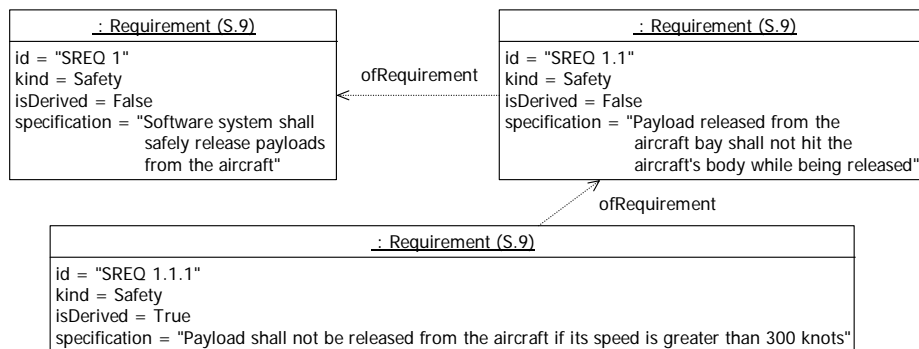


Fig. 11: Decomposing / Refining Safety Requirement and Derived Requirements

It should be noted that Appendix A contains the definitions of concepts, their attributes, and their relationships. Since stereotypes and their tagged values correspond to concepts, their relations, and their attributes, the reader should refer to Appendix A for more detailed descriptions.

Table 4 provides examples of SafeUML stereotypes and tagged values that can be used to capture information valuable for DO-178B. The value of information is evaluated in regards to how it supports verification of compliance with DO-178B requirements. Section 6.5 in the case study provides example uses of the stereotypes in Table 4.

Table 4: Examples of SafeUML stereotypes

<p><i>Section 11.1 bullet c:</i> Project’s Plan for Software Aspects of Certification (PSAC) to include a description of software’s contributions to failure conditions <i>SafeUML:</i> Use stereotype <<SafetyCritical>> to identify software that can contribute to failure conditions, its <i>criticalityLevel</i> tagged value to identify the failure condition level, and its <i>triggersEvent</i> tagged value to identify events that software triggers that may lead to failures</p>
<p><i>Section 11.1 bullet a & Section 11.9 bullet f:</i> Project’s PSAC to include description of hardware/software interfaces in the system, including the requirements of their protocols, frequency of input, and frequency of outputs <i>SafeUML:</i> Use stereotype <<Interface>> to identify interfaces, its <i>isBetweenHardwareAndSoftware</i> tagged value to determine whether it is a hardware/software interface or not, its <i>ProtocolID</i> tagged value to identify the protocol used, its <i>InputFunctionParameter</i> tagged value to identify frequency of inputs, and its <i>OutputFunctionParameter</i> tagged value to identify frequency of outputs</p>
<p><i>Section 11.3 bullet f & Section 11.9 bullet h:</i> Project to specify which methods are used to verify the integrity of partitions performed. Partitioning requirements to be allocated to software, as well as the software level(s) for each partition, be specified <i>SafeUML:</i> Use <<Partition>> stereotype and its tagged values to identify a partition including the partitioning rationale and details, <<SafetyCritical>> stereotype and its tagged values to identify the software level for each partition, and the <<Requirement>> stereotype and its tagged value to identify any partitioning requirements and methods used to verify the integrity of partitions</p>
<p><i>Section 11.1 bullet g:</i> Project’s PSAC to include a description of COTS software used <i>SafeUML:</i> Use <<Nature>> stereotype with its <i>kind</i> tagged value set to COTS, and its <i>reference</i> and <i>explanation</i> tagged values for information and description about the COTS software used</p>
<p><i>Section 11.1 bullet g & Section 11.3 bullet j:</i> Project’s PSAC to include a description of the multiple-version dissimilar software used. A description of the software verification process activities used to verify multiple-version dissimilar software be presented <i>SafeUML:</i> Use <<Replicated>> stereotype and its <i>id</i> tagged value to identify a version with a group of multiple-version dissimilar software, and its <i>replicatedGroupId</i> tagged value to identify the group of multiple-version dissimilar software. Further, use <<Comparator>> stereotype and its tagged value to determine voting parameters. The <<ReliabilityContext>> stereotype is used to identify a group of multiple-version dissimilar software</p>
<p><i>Section 11.7 bullet e:</i> Software design standards to specify which constraints on the software design exist, e.g. recursive software <i>SafeUML:</i> Use stereotype <<ImplementationStyle>> stereotype with its <i>kind</i> tagged value set to Recursive to identify recursive software, and <<Deviation>> stereotype with its <i>kind</i> tagged value set to UsingRecursive to identify recursive software that deviate from design decisions including the rationale</p>
<p><i>Section 11.9:</i> Software requirements to be documented, software design (e.g., UML model) be traced to the software requirements for software assigned level D or above, source code be traceable to the requirements for software assigned level C or above, and design decisions <i>SafeUML:</i> Use <<Requirement>> stereotype and its tagged values to identify and describe a requirement, specifically its <i>isDerived</i> tagged value to indicate whether it is derived or not, and its <i>ofRequirement</i> tagged value to specify the parent requirement to which it traces. The <<Rationale>> stereotype and its tagged values can indicate the rationale for this traceability</p>

Section 11.9 & Section 2.3.3: Specifying safety monitoring requirements, and indication of use as a mechanism for protecting against specific failure conditions by monitoring functions or components

SafeUML: Use <<Event>>, <<Reaction>>, <<Handler>>, and <<Monitor>> stereotypes and their tagged values to identify safety monitoring techniques, events that can impact safety and how they are handled, safety-monitoring software that can identify relevant events, and safety-critical entities that can handle events properly

Finally, it is worth mentioning that reports from model queries do not need to refer to UML terminology and SafeUML stereotypes, but can instead be translated into DO-178B specific terminology (recall that SafeUML stereotypes are traceable to DO-178B concepts). This is important because safety engineers are usually not experienced in UML, object-orientation or programming, and should not have to learn SafeUML terminology.

6. Case Study

In this section, we provide an example of a safety-critical system and illustrate how SafeUML can be used to model it. The goal is to demonstrate, through a realistic case study and a number of representative modeling examples, the usefulness of the SafeUML profile in the context of the identified usage scenarios (see section 2.2) and DO-178B requirements. Stereotypes and tagged values are shown in a number of class diagrams to support the discussion, thereby cluttering a little the diagrams. However, in the context of a modeling tool that would fully support the SafeUML profile, the user would have the option to filter the profile information and associate a graphical notation to various stereotypes, thus avoiding cluttering the class diagram and making the profile key elements clearly and graphically visible.

6.1. System Overview

We considered, as a safety-critical system, the Navigation Controller (NC) subsystem of an aircraft's navigation system (see Fig. 12). This case study was developed only to illustrate the UML profile and, due to confidentiality requirements and for keeping complexity within reasonable bounds, it does not exactly correspond to an actual, certified (e.g., by the FAA) system. However, the first author was involved in the development of similar systems, on which this case study is based, and we followed standard design practices.

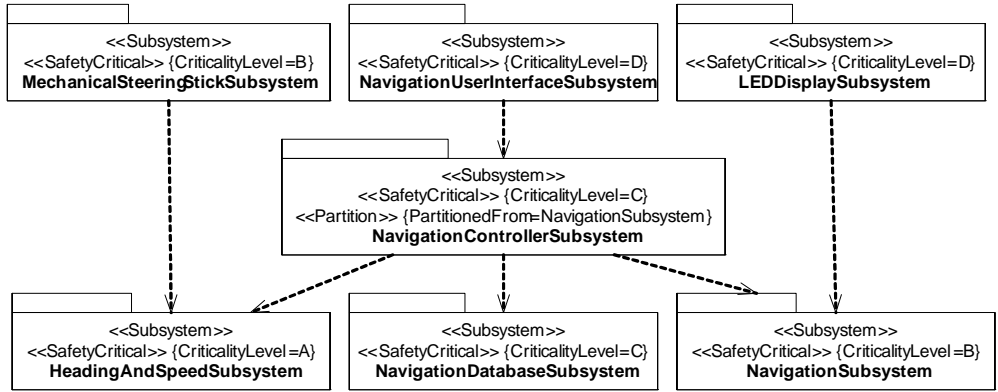


Fig. 12: Software Architecture for the Aircraft Navigation System

The NC subsystem is used to control the aircraft’s flight paths through both automatic pilot and manual input from the pilots. In autopilot mode, the subsystem can choose an appropriate flight path based on the source and destination of the aircraft, and guide the aircraft by generating appropriate commands to the aircraft’s ailerons and spoilers (on the wings), rudder (on the vertical tail), and engines to change the speed and heading (i.e. direction) as required. In custom Fly-To-Point (FTP) mode, the subsystem can accept commands from the pilots such as a destination’s specific latitude and longitude. It then controls the aircraft’s speed and heading to get to the destination the pilot requested.

In order to perform such functionality, the NC subsystem needs to have continuous input from the aircraft’s navigation system, which reports the current position and altitude of the aircraft at all times. In addition, it needs to be able to command the aircraft’s ailerons, spoilers, rudders, and engines to change the speed and heading.

To develop the subsystem, we first identified five high-level functional requirements (main functionalities) for the subsystem, referred to as **FREQ 1** to **FREQ 5**. We then performed a safety assessment, using four standard, complementary methods (namely Action Error Analysis, Failure Modes and Effects Analysis, Hazards and Operability Analysis, and Interface Analysis [11]). This generated eleven safety requirements, referred to as **SREQ 1** to **SREQ 7** (**SREQ 6** is further decomposed into **SREQ 6.1** to **SREQ 6.5**). We then designed the subsystem while recording design decisions (in particular safety-related ones) using our profile. Lastly, we evaluated the resulting design documents with respect to the usage scenarios discussed in Section 2.2.

Below we detail the development of the NC subsystem in four steps. We first identify and list the functional and safety requirements of the NC subsystem in section 6.2. Second, we identify the events the controller subsystem has to handle, and the reactions it has to take, according to the functional and safety requirements (Section 6.3). Third, we discuss the design of the subsystem itself (Section 6.4) and we record design decisions using the SafeUML profile. Last, we show how safety engineers and third party certification authorities can use this information (Section 6.5).

6.2. Functional and Safety Requirements

The subsystem's functional and safety requirements are described in Table 5. We list here the requirements assuming that they have already gone through sufficient refinement and decomposition, and that derived requirements have already been identified and added. In section 5.2 we showed how documenting them is possible using SafeUML.

Table 5: Functional and safety requirements for the Navigation Controller (NC) subsystem

Number	Requirement
FREQ 1	NavigationControllerSubsystem shall be able to list pre-determined flight paths for a requested source/destination pair
FREQ 2	NavigationControllerSubsystem shall provide an autopilot feature where it flies the aircraft through a requested flight path
FREQ 3	NavigationControllerSubsystem shall be able to fly the aircraft to a requested Fly-To-Point (FTP)
FREQ 4	NavigationControllerSubsystem shall provide the capability to guide the pilots through a requested flight path when the pilot is controlling the aircraft through MechanicalSteeringStickSubsystem
FREQ 5	NavigationControllerSubsystem shall be able to provide navigation information received from NavigationSubsystem
SREQ 1	NavigationControllerSubsystem shall disable autopilot and FTP features when the pilot is using MechanicalSteeringStickSubsystem, and re-enable them when the pilot stops using MechanicalSteeringStickSubsystem
SREQ 2	NavigationControllerSubsystem shall be able to identify whether a specific LAT/LONG position is in a safe area or not, and not fly the aircraft to unsafe positions unless explicitly confirmed by the pilot
SREQ 3	NavigationControllerSubsystem shall be able to determine whether flying to a specific LAT/LONG position requires flying through unsafe areas or not, and not fly the aircraft through unsafe areas unless explicitly confirmed by the pilot
SREQ 4	NavigationControllerSubsystem shall alert the pilot when the next FTP cannot be reached without having to refuel the aircraft
SREQ 5	When NavigationControllerSubsystem fails, an alert shall be raised and, until NavigationControllerSubsystem is operational again, the pilot shall be required to manually fly the aircraft using MechanicalSteeringStickSubsystem
SREQ 6	NavigationControllerSubsystem shall ensure that the autopilot and FTP features are enabled only when all of the following conditions hold
SREQ 6.1	WingsAndEnginesSubsystem is functional
SREQ 6.2	NavigationDatabaseSubsystem is functional
SREQ 6.3	NavigationSubsystem is functional
SREQ 6.4	NavigationControllerSubsystem is able to communicate with WingsAndEnginesSubsystem
SREQ 6.5	NavigationControllerSubsystem is able to communicate with NavigationSubsystem
SREQ 7	NavigationControllerSubsystem shall require explicit confirmation to continue autopilot or FTP flight modes every 5 minutes until NavigationSubsystem indicates that the GPS feature is functional again. If the confirmation is not performed for a period of 7 consecutive minutes, then NavigationControllerSubsystem shall signal an emergency to the pilots

6.3. Identification of Events and Reactions

A standard practice is to identify all the events the NC subsystem receives, and the reactions it performs, which could have safety implications. This advocates an Event-Driven Architecture (EDA). To identify the events, one needs to determine which inputs to the system, or changes in its state, may

impact its safety. To identify the reactions, one needs to determine how the system should behave to ensure safety when any of the identified events occurs.

Safety requirements can be used to identify such safety-related events and reactions. For example, one can identify at least two events of interest from safety requirement SREQ 1 above: (1) The event of when the pilot starts using subsystem `MechanicalSteeringStickSubsystem`; (2) The event of when the pilot stops using this subsystem. Also from this requirement, one can identify at least the following reactions: (1) Disabling the autopilot when the pilot starts using the mechanical and steering stick subsystem; (2) Enabling the autopilot when the pilot stops using this subsystem.

Such a requirements analysis led to the identification of an inheritance hierarchy of event classes (stereotyped `<<Event>>` (S.21)) and an inheritance hierarchy of reaction classes (stereotyped `<<Reaction>>` (S.22)). Excerpts of the two hierarchies appear in Fig. 13, showing the events and reactions identified from SREQ 1, and the complete hierarchies can be found in [24]. The `<<Event>>` (S.21) hierarchy shows two concrete classes, namely `HeadingAndSpeedControlledByOtherSubsystem` (raised when pilot starts using subsystem `MechanicalSteeringStickSubsystem`) and `HeadingAndSpeedNotControlledByOtherSubsystem` (raised when pilot stops using subsystem `MechanicalSteeringStickSubsystem`). The `<<Event>>` (S.21) stereotype has two tagged values: one provides information on the impact of system safety due to the subsystem's receipt of the event (`EffectOnSafetyDirection`), and the other provides some context under which the event can be raised (`EffectOnSafetyContext`).

The diagram also shows two concrete reaction classes, namely `DisableController` and `EnableController`, which are stereotyped with `<<Reaction>>` (S.22). The `<<Reaction>>` (S.22) stereotype has a number of tagged values to: refer to the event(s) that the reaction is a consequence of (`ConsequenceOf`), whether it increases (`Positive`) or decreases (`Negative`) the level of software system safety (`EffectOnSafetyDirection`), and indicate the condition(s) under which the reaction

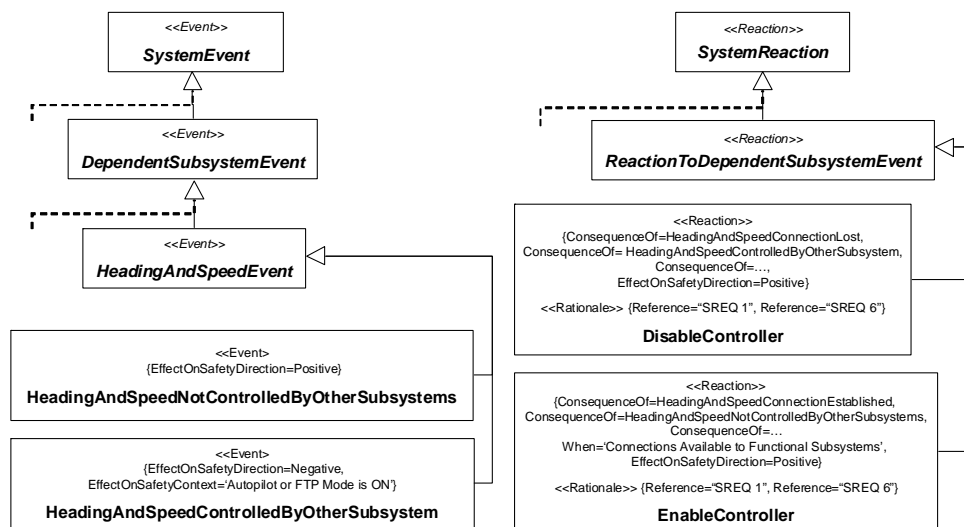


Fig. 13: Excerpt of the `<<Event>>` and `<<Reaction>>` hierarchies

occurs (*When*). For example, when connections to the relevant subsystems are restored, the `NavigationControllerSubsystem` is re-enabled and it resumes controlling them. This is documented in Fig. 13 through the `EnableController` reaction, which can be triggered by the `HeadingAndSpeedNotControlledByOtherSubsystems` event (tag `ConsequenceOf`) when *connections are available to functional subsystem* (tag `When`), and this has a *positive* effect on system safety (tag `EffectOnSafetyDirection`) (in addition, it is possible to use the `EffectOnSafetyValue` tag to quantify the change in safety). Fig. 13 also illustrates that reactions `DisableController` and `EnableController` are also consequences of events other than the one previously mentioned (e.g., `HeadingAndSpeedConnection Established`).

The two `<<Reaction>>` (S.22) classes are also stereotyped `<<Rationale>>` (S.14) to record the design decisions for including them. The `<<Rationale>>` (S.14) stereotype has a tagged value, namely `Reference`, to refer to the requirements that justify the existence of the reactions (here, safety requirements SREQ 1 and SREQ 6).

6.4. Controller Subsystem Design – Usage Scenarios 1, 2, 3

The design (class diagram) of the controller subsystem appears in Fig. 14, and selected stereotypes are applied to the model for illustrative purposes. The diagram (i.e., package) itself is stereotyped `<<SafetyContext>>` (S.16) to specify that it contains information that is relevant to system safety. It is also stereotyped `<<Requirement>>` (S.9) to indicate the functional and safety requirements that are relevant to the diagram (tagged values `Kind` and `Specification`). In this case, the stereotypes indicate that the design fulfils all requirements.

Class `Controller` is also stereotyped with `<<Handler>>` (S.23) with tagged value `HandleableEvent` equal to `PilotInputEvent` to indicate that it handles all concrete events that are subclasses of class `PilotInputEvent` (stereotyped `<<Event>>` (S.21), but not shown in Fig. 13). In addition to functional features (e.g., changing the flight path in response to a `ChangeFlightPath` input event), `Controller` also executes safety features such as the `InvestigateFuelShortage` reaction (tagged value of `<<Handler>>`) to determine whether the changes requested by the pilots may result in a fuel shortage or not.

Class `Controller` is also stereotyped `<<state dependent control>>`, indicating it is a control class with a state-dependent behaviour. This stereotype, as well as `<<algorithm>>`, `<<coordinator>>`, and `<<system interface>>` are not part of the SafeUML profile. They were defined by the Concurrent Object Modeling and architectural design mETHOD (COMET) methodology for Real-time systems [3] as a strategy to distribute responsibility across classes.

The controller subsystem interacts with the HeadingAndSpeedSubsystem, the NavigationDatabaseSubsystem, and the NavigationSubsystem (Fig. 12). Therefore, the Controller class is associated with three <<system interface>> classes (following COMET recommendations [3]), namely HeadingAndSpeedInterface, NavigationDatabaseInterface, and NavigationInterface. These interface classes are stereotyped with <<Rationale>> (S.14), thereby justifying the design decision to include them by relating them to functional and safety requirements. They are also stereotyped <<Interface>> (S.20) to indicate the kind of interface they correspond to (i.e., software-software, or software-hardware). (Note that identifying the software-hardware interfaces is a requirement of the DO-178B [22].) Class Controller can receive events from those subsystems (through the corresponding interface classes). These events are monitored by three <<Monitor>> (S.24) classes (i.e., HeadingAndSpeedMonitor, NavigationDatabaseMonitor, Navigation-Monitor), which all interact with one handler to trigger the required reactions, namely ExternalSubsystemsEventHandler (bottom of Fig. 14). The <<Monitor>> (S.24) stereotype

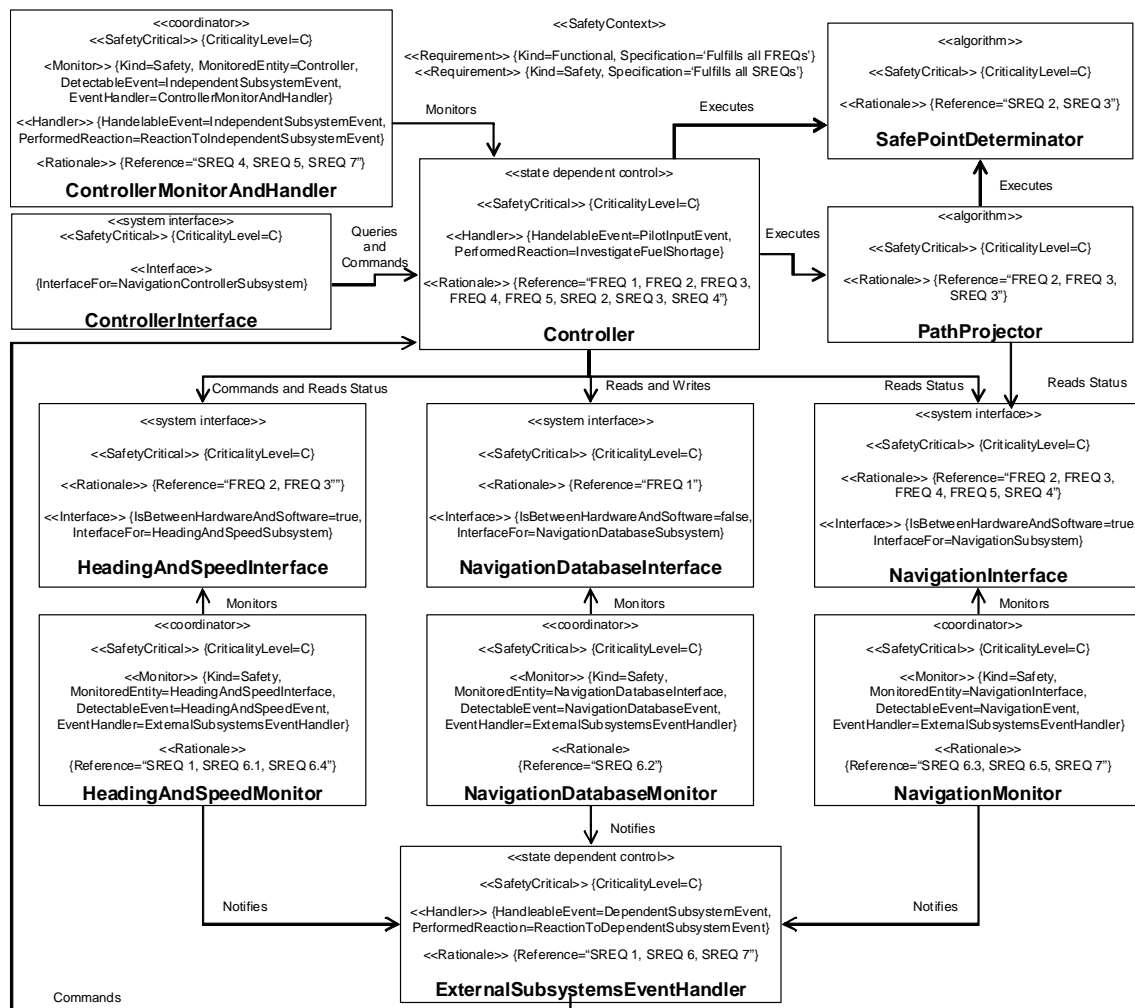


Fig. 14: Controller Subsystem Design

indicates that those monitors are safety monitors (`Kind` tagged value is set to `Safety`). It also specifies the monitored entity (tagged value `MonitoredEntity`), the detectable event from this entity (tagged value `DetectableEvent`), and the handler for those events (tagged value `EventHandler`). In particular, referring to the already discussed safety requirement SREQ 1, class `HeadingAndSpeedMonitor` monitors class `HeadingAndSpeedInterface` to detect `HeadingAndSpeedEvent` (tagged value `DetectableEvent`), and therefore its event subclasses `HeadingAndSpeedControlledByOtherSubsystem` and `HeadingAndSpeedNotControlledByOtherSubsystem` events (Fig. 9), and then notifies `ExternalSubsystemsEventHandler` that triggers the `DisableController` or `EnableController` reactions accordingly. (The `PerformedReaction` tagged value of the `<<Handler>>` (S.23)'s stereotype indicates `ReactionToDependentSubsystemEvent`, which `DisableController` and `EnableController` inherit from—Fig. 13). Classes `HeadingAndSpeedMonitor` and `ExternalSubsystemsEventHandler` are therefore stereotyped `<<Rationale>>` (S.14), and the `Reference` tagged value shows SREQ 1.

While explaining the subsystem design in this section, we provided numerous examples of using SafeUML to communicate safety information to software engineers (Usage 1), designing safety into the software (Usage 2), and documenting justifications for design decisions (Usage 3). This is apparent through the use of SafeUML stereotypes in Fig. 12 and Fig. 14, and the extensive explanation provided here. The next section will illustrate how such a SafeUML design model in this case study can support safety engineering and certification by providing case study examples on information required for safety monitoring (Usage 4) and certification (Usage 5), thus also illustrating SafeUML benefits to safety engineers.

6.5. Safety Analysis and Certification based on Design Models – Usage Scenarios 4, 5

In this section, we illustrate, using the NC subsystem case study, how Usage 4 and Usage 5 discussed in Section 2.2 (Fig. 1) can be facilitated through the use of SafeUML models. In section 5.2, we extensively discussed the benefits of modeling using SafeUML for DO-178B and safety monitoring, and we presented several examples of DO-178B guidelines and how using SafeUML helps engineers satisfy them. In the previous section, we presented concrete examples from the SafeUML model in this case study (Fig. 12 and Fig. 14) to illustrate the benefits for Usage 1, Usage 2, and Usage 3. In this section, we provide examples on the benefits for Usage 4 and Usage 5.

Recall that a project's PSAC (Plan for Software Aspects of Certification) is required to include a description of software's contributions to failure conditions (see section 5.2). This is in essence software levels, which can be obtained by querying the model for all the model elements with the `<<SafetyCritical>>` stereotype and reading their `CriticalityLevel` tagged value.

Further, the software level for each component or element must be as high as the highest level for all elements that depend on it. From Fig. 12, we can extract a list of safety-critical elements and

failure condition category level (using stereotype <<SafetyCritical>>). Notice that the software level for each element is at least equal to the highest level for all elements that depend on it. This is summarized in Table 6. This allows safety engineers and certification authorities (Usage 4 and Usage 5) to ensure that DO-178B guidelines with respect to software level design rules were followed in Usage 2 and Usage 3. For example (Table 6), `NavigationSubsystem` has level B, which is higher than the levels for both elements that depend on it, namely `NavigationControllerSubsystem` (level C) and `LEDDisplaySubsystem` (level D).

Table 6: Software levels of components and the components that depend on them

Safety-Critical Element (Level)	Elements Dependent on It (Level)	Notes
HeadingAndSpeedSubsystem (A)	MechanicalSteeringStickSubsystem (B) NavigationControllerSubsystem (C)	$A \geq B$ $A \geq C$
NavigationDatabaseSubsystem (C)	NavigationControllerSubsystem (C)	$C \geq C$
NavigationSubsystem (B)	NavigationControllerSubsystem (C) LEDDisplaySubsystem (D)	$B \geq C$ $B \geq D$
NavigationControllerSubsystem (C)	NavigationUserInterfaceSubsystem (D)	$C \geq D$

Recall that a project’s PSAC is required to include a description of the hardware/software interfaces in the system (section 5.2): DO-178B places special emphasis on this, especially on interfaces between hardware and software. From Fig. 14, we can extract such a list, using stereotype <<Interface>>, to support Usage 4 and Usage 5, and identify hardware/software interfaces if they were documented in Usage 3 (Table 7). `NavigationInterface`, for examples, is a software class to interface with the `NavigationSubsystem` hardware subsystem (Table 7).

Table 7: Examples of component interfaces (including hardware/software ones)

Interface	To Entity	Between Hardware and Software
ControllerInterface	NavigationControllerSubsystem	No
HeadingAndSpeedInterface	HeadingAndSpeedSubsystem	Yes
NavigationDatabaseInterface	NavigationDatabaseSubsystem	No
NavigationInterace	NavigationSubsystem	Yes

Recall that a project’s PSAC is required to include a documentation of requirements and their traceability through design (section 5.2). From Fig. 14, we can extract such a list. This will help ensure that all safety requirements are designed through the software (through stereotypes <<Requirement>>, <<Rationale>>). Table 8, generated from the SafeUML model, helps the safety engineer to conclude, while monitoring safety (Usage 4), that all safety requirements are addressed by some design. Further, this traceability information can be submitted to certification authorities (Usage 5) to illustrate requirements traceability from requirements to design (i.e. which design elements, with corresponding generated code, trace to which requirements). For example, from Table 8 we can conclude that `HeadingAndSpeedMonitor` and `ExternalSubsystemsEventHandler` implement SREQ 1 and that all functional and safety requirements are designed into the system. If some requirements

were not designed into any design element, then safety engineers can raise a flag and software engineers can further investigate to resolve this.

Table 8: Traceability between requirements and design

Design Element \ FREQ → Functional SREQ → Safety	F	F	F	F	F	S	S	S	S	S	S	S	S	S	S	S
	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
	Q	Q	Q	Q	G	Q	Q	Q	Q	Q	Q	Q	Q	Q	Q	Q
	1	2	3	4	5	1	2	3	4	5	6.1	6.2	6.3	6.4	6.5	7
Entire diagram	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
ControllerMonitorAndHandler									√	√						√
Controller	√	√	√	√	√		√	√	√							
SafePointDeterminator							√	√								
PathProjector		√	√					√								
HeadingAndSpeedInterface		√	√													
NavigationDatabaseInterface	√															
NavigationInterface		√	√	√	√				√							
HeadingAndSpeedMonitor						√					√			√		
NavigationDatabaseMonitor												√				
NavigationMonitor													√		√	√
ExternalSubsystemsEventHandler						√					√	√	√	√	√	√

In order to help safety engineers monitor safety (Usage 4), it is possible to extract a list of safety-critical events and how they will be handled (using stereotypes <<Event>>, <<Reaction>>, <<Monitor>>, <<Handler>>). This information can be extracted from Fig. 14, as summarized in Table 9 that lists events, reactions, monitors, handlers, and classes that raise events. Notice that Table 9 does not specify which design elements raise and detect the `PilotInputEvent` event (empty cells), which suggests that software engineers did not record and justify some relevant decisions for the corresponding safety requirements or reactions (missing activity in Usage 3). The cause of this may be that software engineers did not design safety requirements in software (missing activity in Usage 2).

Table 9: Safety critical events and handling mechanisms

Event	Raised By	Detected By	Handled By	Reaction
Independent SubsystemEvent	Controller	ControllerMonitor AndHandler	ControllerMonitor AndHandler	ReactionTo Independent SubsystemEvent
PilotInputEvent	-	-	Controller	InvestigateFuelShortage
HeadingAnd SpeedEvent	HeadingAnd SpeedInterface	HeadingAnd SpeedMonitor	ExternalSubsystem EventHandler	ReactionToDependent SubsystemEvent
Navigation DatabaseEvent	Navigation DatabaseInterface	Navigation DatabaseMonitor	ExternalSubsystem EventHandler	ReactionToDependent SubsystemEvent
Navigation Event	Navigation Interface	Navigation Monitor	ExternalSubsystem EventHandler	ReactionToDependent SubsystemEvent

7. Conclusion

Communication and collaboration among safety stakeholders, namely safety engineers, software engineers, and certification authorities, was found to be a major challenge when developing software for safety-critical systems such as airborne systems [5]. To help resolve those challenges, we focused our research on the civil and military aerospace systems development using RTCA DO-178B [22]. We first identified and modeled, from the DO-178B [22] standard, the information that both engineering groups have to exchange and how they would use such information in practice (usage scenarios): safety engineers communicate safety requirements to software engineers, software engineers design software with system safety in mind and record design decisions, safety engineers monitor system safety during design and prepare reports for certification. We modeled this information as a conceptual model with 26 safety-related concepts, 48 attributes, and 35 relationships between concepts. Although primarily focusing on developing software for the aerospace industry, those concepts would also be reusable for other safety-critical industries.

To embed safety information in UML models, the de-facto standard for modeling object-oriented software, we defined information requirements from the conceptual model. We then evaluated the suitability of existing UML-based solutions (e.g., existing profiles). Our conclusion was that existing UML-based solutions are far from being adequate for our purpose, as they only capture a small portion of the safety concepts in our conceptual model. We therefore defined our own UML profile, composed of 31 stereotypes and 79 tagged values, focusing first on adding safety information to class diagrams.

This profile was applied on a realistic case study, an aircraft's navigation controller subsystem that can control the aircraft's flight paths through both automatic pilot and manual input from the pilots. We designed the controller subsystem using the SafeUML profile, i.e., embedding safety information in the model, following guidance specified in the DO-178B standard (e.g., we performed a safety analysis of the controller subsystem). We then showed that the SafeUML profile can indeed facilitate communication among software engineers, safety engineers, and certification authorities.

Future work should first focus on (1) providing full automation for the profile including defining a list of common model queries required for certification under the DO-178B standard, (2) extending it to other diagrams such as sequence diagrams and state machines to help analyse the safety behavioural aspects of the systems (e.g., sequences of events and reactions), and (3) further evaluating its cost and effectiveness in practice through additional case studies.

Acknowledgements. This work was performed within the framework of Gregory Zoughbi's Master's thesis. Gregory Zoughbi was partly supported by General Dynamics Canada employee educational tuition assistance. This work was partly supported by a Canada research Chair (CRC) grant. Lionel Briand and Yvan Labiche were further supported by NSERC discovery grants.

References

- [1] Balasubramanian K., Krishna A. S., Turkay E., Balasubramanian J., Parsons J., Gokhale A., Schmidt D., "Applying model-driven development to distributed real-time and embedded avionics systems", *Proc. of International Journal of Embedded Systems - Vol. 2, No.3/4 pp. 142 - 155*, 2006
- [2] CENELEC EN 50128: Railway Applications: Software for Railway Control and Protection Systems, Version 1997
- [3] Gomaa H., "Designing Concurrent, Distributed, and Real-Time Applications with UML," Object Technology, Addison Wesley, 2000.
- [4] Hansen K. T. and Gullesen I., "Utilizing UML and Patterns for Safety Critical Systems," *Proc. Workshop on Critical Systems Development with UML, in conjunction with the International Conference on the UML*, 2002.
- [5] Hayhurst K. J. and Holloway C. M., "Challenges in Software Aspects of Aerospace Systems," *Proc. Annual NASA Goddard Software Engineering Workshop*, 2001.
- [6] Herrmann D. S., *Software Safety and Reliability: Techniques, Approaches, and Standards of Key Industrial Sectors*, Wiley, 2000.
- [7] IBM, Eclipse Modeling Framework, <http://www.eclipse.org/modeling/emf>, August 2009
- [8] International Electrotechnical Commission (IEC), "Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems," IEC 61508, 1998.
- [9] Jürjens J., "Developing Safety-Critical Systems with UML," *Proc. International Conference on the UML*, LNCS 2863, pp. 360-372, 2003.
- [10] Lagarde F., Espinoza H., Terrier F., Andre C. and Gerard S., "Leveraging Patterns on Domain Models to Improve UML Profile Definition," *Proc. Fundamental Approaches to Software Engineering*, LNCS 4961, pp. 116-130, 2008.
- [11] Leveson N. G., *Safeware - System Safety and Computers*, Addison-Wesley, 1995.
- [12] Lewis R., Dale C., Anderson T., "Safety Case Development as an Information Modelling Problem," Editors Safety-Critical Systems: Problems, Process and Practice, *Proc. of the Seventeenth Safety-Critical Systems Symposium*, February 2009.
- [13] Meunier J.-N., Lippert F. and Jadhav R., "RT Modeling with UML for Safety Critical Applications - the HIDOORS Project Example," *Proc. Workshop on Specification and*

Validation of UML Models for Real-Time and Embedded Systems, in conjunction with the International Conference on the UML, 2003.

- [14] Nilsen K., "Certification Requirements for Safety-Critical Software," *RTC Magazine*, 2004.
- [15] OMG, "A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems," Version 1.0 Beta 2, <http://www.omg.org/docs/ptc/08-06-08.pdf>, 2008.
- [16] OMG, "Object Constraint Language", <http://www.omg.org/docs/formal/06-05-01.pdf>, May 2006.
- [17] OMG, "UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms," Final Adopted Submission, <http://www.omg.org/docs/ptc/05-05-02.pdf>, 2005.
- [18] OMG, "UML Profile for Schedulability, Performance, and Time Specification," Adopted Specification, <http://www.omg.org/docs/formal/05-01-02.pdf>, 2005.
- [19] OMG, "Unified Modeling Language: Infrastructure," <http://www.omg.org/docs/formal/07-11-04.pdf>, November 2007.
- [20] OMG, "Unified Modeling Language: Superstructure," <http://www.omg.org/docs/formal/07-11-02.pdf>, November 2007.
- [21] Pender T., *UML Bible*, Wiley, 2003.
- [22] RTCA, "Software Considerations in Airborne Systems and Equipment Certification," Radio Technical Commission for Aeronautics (RTCA), European Organization for Civil Aviation Electronics (EUROCAE), Standard Document no. DO-178B/ED-12B, December, 1992.
- [23] Zoughbi G., Briand L. C. and Labiche Y., "A UML Profile For Developing Airworthiness-Compliant (RTCA DO-178B) Safety-Critical Software," *Proc. Model Driven Engineering Languages and Systems 2007*, LNCS 4735, pp. 574-588, 2007.
- [24] Zoughbi G., Briand L. C. and Labiche Y., "A UML Profile For Developing Airworthiness-Compliant (RTCA DO-178B) Safety-Critical Software," Carleton University, Technical Report SCE-05-19, December, 2006.

Appendix A – Safety Conceptual Model

Table 10 lists and explains each safety-related concept in Fig. 2, while Table 11 lists and explains the relationships across those safety-related concepts.

Table 10: Safety-related concepts—Definitions

<p>(C.1) Requirement: specifies a requirement that must be met. The requirement need not necessarily be a safety requirement – it can be any functional or non-functional requirements. It may be traceable to another requirement, which is often a higher level one. This enables the concept of requirements traceability, which is a key element in the software development process</p> <p><i>id:</i> a unique ID for this requirement. Examples: “REQ 1”, “REQ 2”, “FREQ 1”, “SREQ 10”, ... etc</p> <p><i>is derived:</i> indicates this requirement is derived from another one or not. Examples: “True”, “False”</p> <p><i>kind:</i> the kind of this requirement. Examples: “Functional”, “Safety”, “Reliability”, “Integrity”, “Performance”, “Concurrency”, “Certification”, “Design”, “Configuration”, ... etc</p> <p><i>specification:</i> the actual requirement’s specification. Examples: “Radar Output is Poisson with Lambda = 20 ms”, “Levels of Code Nesting < 5”, ... etc</p>
<p>(C.2) Deviation: identifies a design deviation from a plan, standard, or Requirement (C.1). Deviations are important to note as they must be submitted to the certification authorities according to the DO-178B standard [22].</p> <p><i>kind:</i> the kind of this deviation. This generally specifies the deviation action or decision. Examples: “Using Recursive Algorithm”, “Using Dynamic Memory”, ... etc</p> <p><i>explanation:</i> specifies how and why, this is a deviation from the reference requirements (see relationships to other concepts). Examples: “Kalman filter is recursive so using recursive algorithm for the implementation”, ... etc</p>
<p>(C.3) Style: an abstract concept indicating an implementation or a behavioural style. It does not capture any information, but it serves as a base class for other concepts</p>
<p>(C.4) ImplementationStyle: identifies a style that is used to implement a design. A development standard should define which styles are permitted and which ones are not, and this concept identifies conformance or deviation from the standard’s requirements</p> <p><i>kind:</i> the kind of this implementation style. Examples: “Recursive”, “Unbounded Loop”, “Compacted Expression”, “Dynamic Memory”, “Data Alias”, ... etc</p> <p><i>parameters:</i> describes additional details of the implementation style. It is generally an expression whose meaning is dependent on the Kind of the implementation style. Examples: “Dynamic memory allocation frequency = Poisson with Lambda = 15 seconds”, ... etc</p> <p><i>explanation:</i> specifies how this implementation style conforms to, or deviates from, the reference requirements (see relationships to other concepts). Examples: “Using dynamic memory here because static because 90% of the time only 10% of the maximum memory space will be needed (which would be required if static memory is used). This improves performance”, ... etc</p>
<p>(C.5) BehaviouralStyle: identifies and describes a behavioural style of a design. A development standard should define which styles are permitted and which ones or not, and this concept identifies conformance or deviation from the standard’s requirements</p> <p><i>kind:</i> The kind of this behavioural style. Examples: “Time-Related”, “State-Related”, ... etc</p> <p><i>parameters:</i> describes additional details of the behavioural style. It is generally an expression whose meaning is dependent on the Kind of the behavioural style. Examples: “Number of state machine states = 10”, “Number of state transitions = 20”, “Frequency of state changes = Periodic every 1 minute”, ... etc</p> <p><i>explanation:</i> specifies how this behavioural style conforms to, or deviates from, the reference requirements (see relationships to other concepts). Examples: “Frequency of state changes is less than the maximum value permitted by REQ 23”, ... etc</p>
<p>(C.6) Nature: describes the source for the design such as whether the actual software is purchased to meet the requirements, whether it was previously developed as part of another project or software system, or whether it is deactivated and does not get executed</p> <p><i>kind:</i> the kind of the software’s nature. It is the primary attribute that describes the actual software represented by this concept. Examples: “COTS”, “Deactivated”, “Previously Developed”, ... etc</p> <p><i>explanation:</i> specifies how the referenced requirements are met by the nature of this design (see relationships to</p>

other concepts). Examples: “This is a COTS software component purchased according to document number 1234567 to meet requirements REQ 1 – REQ 10”, ... etc

(C.7) Rationale: specifies that a specific design exists to support another design element, or to fulfill specific requirements. It explicitly allows modelers to trace the design to specific Requirements (C.1)

explanation: specifies how the design decision is a solution for the referenced requirements. Examples: “This class lists safe flight paths for an aircraft, which is used to satisfy safety requirements SREQ 1, SREQ 2, and SREQ 3”, ... etc

(C.14) Event: describes an event or action that may occur. An event may impact system safety by either causing or removing hazards. It may also be caused internally by the system or it may be an external event. It does not need another event to trigger it

kind: the kind of this event. Examples: “External”, “Internal”,... etc

when: describes the conditions under which this event occurs. This may be specified in a formal language.

Examples: “Event occurs when a sonobuoy is released from the aircraft”, ... etc

effect on safety direction: specifies the direction of its impact on system safety, i.e. whether it removes some hazards, does not impact safety, or causes additional hazards to occur. Therefore, this attribute provides qualitative information. Examples: “Positive”, “Neutral”, “Negative”, ...etc

effect on safety value: specifies the severity of its impact on system safety. This is also used to quantify the impact on safety, possibly by identifying the effect of the event on the number of hazards in the system.

Therefore, this attribute provides quantitative information. Examples: “+5”, “0”, “-5”, ... etc

effect on safety context: identifies the context within which the “Effect On Safety Direction” and “Effect On Safety Value” attributes are valid. This attribute is necessary because understanding the context is essential to safety [2]. Examples: “Aircraft is flying above water”, “Aircraft is on the ground”, “Aircraft is in autopilot mode”, ...etc

(C.15) Reaction: describes a response/reaction to one or more Events (C.14) that may occur. A reaction may impact system safety by either causing or removing hazards. It is an Event (C.14) in itself, but it always occurs in response to other Events (C.14). It is a subclass of the Event (C.14) concept to allow the possibility of chain reactions (i.e. there could be a reaction for a reaction)

kind: inherited from “Event” (C.14)

when: inherited from “Event” (C.14) . In effect, this attribute filters out situations when the reaction will not be performed as a result of the Event (C.14) occurrence. This may be specified in a formal language

effect on safety direction: inherited from “Event” (C.14)

effect on safety value: inherited from “Event” (C.14)

effect on safety context: inherited from “Event” (C.14)

(C.9) SafetyCritical: represents a safety-critical design element that impacts system safety. It also identifies the safety or airworthiness level of design elements (within the system in context)

criticality level: indicates the level of criticality (e.g. airworthiness level, Safety Integrity Level (SIL)), on some pre-defined scale, such as the software level or the failure condition category. Examples: For RTCA DO-178B [5]: “A”, “B”, “C”, “D”, “E”. For IEC 61508 [20]: “SIL 1”, “SIL 2”, “SIL 3”, “SIL 4”. For ... etc

confidence level: indicates the level of confidence, on some pre-defined scale, that the criticality level is satisfied. Examples: “High”, “Medium”, “Low”, “80%”, “50%”, ... etc

(C.8) Partition: identifies a design partition that resulted from separating some design element from other design elements. Partitioning is a technique for providing isolation between functionally independent entities to contain and/or isolate faults and potentially reduce the effort of the verification process. It prevents specific interactions and cross-coupling interference [2]. Its key advantages are in separating safety-critical design elements that have different safety levels, so that the failure of the less critical entity does not result in the failure of the more critical entities

explanation: provides further details on the reasons for the partitioning. Examples: “Partitioned away from a software component with a higher airworthiness level”, ... etc

(C.16) Handler: identifies a design element that handles Events (C.14) that are detected by a Monitor (C.17). A handler handles the Events (C.14) by performing specific Reactions (C.15) in response to the Events (C.14)

(C.17) Monitor: identifies a design element that monitors other Safety-Critical (C.9) design elements for Events (C.14). Detected Events (C.14) are passed to Handlers (C.16) for processing, which in turn invoke the appropriate Reactions (C.15)

kind: the kind of this monitor, indicating the quality of service that it monitors. Examples: “Safety”, “Reliability”, “Integrity”, “Performance”, “Concurrency”, “Configuration”,... etc

(C.10) EnvironmentModel: identifies a design element that models a SafetyCritical element by mimicking the behaviour, usually in test mode, of another design element that will be used in the real system. For example, software simulators are instances of EnvironmentModels, and it is common to use them to model hardware elements or other subsystems (hardware or software) for system integration and testing purposes. EnvironmentModels are often used in developing large systems, they make the testing experience easier and more cost effective, and they play a key role in System Integration Labs (SIL) [2]

parameters: specifies which behaviours are modeled and how. Examples: for a communication subsystem model/simulator (e.g. Radio Frequency (RF)): “Messages received as Poisson with Lambda = 100ms”, “Message loss frequency is Poisson with Lambda = 250 messages”, ... etc

(C.11) Strategy: describes an approach used to in a specific design element. This approach could be a design decision that relates to some category (see *kind* attribute below)

kind: the kind of this strategy. Examples: “Safety”, “Reliability”, “Integrity”, “Performance”, “Concurrency”, “Certification”, “Design”, “Configuration”, “Scheduling”, “Formalism”... etc

parameters: specifies the strategy policy parameters. Examples: for a Scheduling strategy: “Round Robin”, “FIFO”, “LIFO”, ... etc, for a Formalism strategy (i.e. use of formal methods): “Natural Deduction”, “Linear Logical Framework (LLF)”, ... etc

(C.12) Measure: quantifies a characteristic of a design element. Measures can be quantified for various types of metrics such as coupling between entities, complexity of a single entity, cohesion of a single entity, size, ... etc

kind: identifies the kind of the measurement that is used to quantify the SafetyCritical entity. Examples: “Level of Nested Calls”, “Conditional Structures”, “Unconditional Branches”, “Number of Entry/Exit Points of Code”, “Big O”, “Source Lines of Code”, “Inter-Entity Call Points”, ... etc

value: an expression specifying the value, or the permitted range, of the measure. Examples: “n2”, “log n”, “25”, ... etc

(C.13) Interface: describes an interface between design elements. Interfaces are common between subsystems of the same system, between the system and some other external system, between software and hardware, and other situations

is between hardware and software: indicates whether the interface is between hardware and software. Examples: “True”, “False”

protocol id: identifies the protocol used. Examples: “MIL STD 1553” [21], “Ethernet”, “CORBA”, ... etc

input function parameters: specifies the expected input function and/or its frequency. Examples: “Poisson with Lambda = 20ms”, “Periodic every 1 second”, ... etc

output function parameters: specifies the expected output function and/or its frequency. Examples: “Poisson with Lambda = 20ms”, “Periodic every 1 second”, ... etc

(C.18) Concurrent: identifies a design element that participates in a concurrency model. There are several possible roles that the design element can assume in a concurrency model, such as being a resource or software execution code that can be either active or passive. An active design element is one that is capable of generating stimuli concurrently or pseudo (seemingly) concurrently without being prompted by an explicit stimulus instance, whereas a passive one is one that cannot generate its own behaviour but only reacts when prompted by a stimulus [8]

role: the role of this entity. Examples: “Active”, “Passive”, “Resource”

is shared: specifies whether this entity can be shared by more than one other entity or not. Examples: “True”, “False”

parameters: specifies how this entity acts from a concurrency point of view, such as the frequency of events that an active entity can trigger, or the maximum frequency at which a passive entity or a resource can be accessed. Examples: “Poisson with Lambda = 20ms”, “Periodic every 1 second”, ... etc

(C.19) Defensive: specifies that a design element employs a defensive design model, and describes it. In a defensive design model (e.g. defensive programming model for software), a design element checks for illegal inputs and forbid execution using illegal inputs, thus avoiding a scenario where the design element may fail due to an unfulfilled assumption on the input variables

defendable inputs: specifies illegal input conditions that this design element checks against. Examples: “Division by Zero”, “Altitude < 0”, ... etc

(C.20) Configuration: represents a specific configuration. Software and/or hardware configurations may change by changing memory bits, changing lookup tables, loading a software patch, and others

id: uniquely identifies a specific software configuration. Examples: for a user interface software that can provide interface in many languages based on a string lookup table: “English Interface”, “French Interface”,

<p>“German Interface”, ... etc</p>
<p>(C.21) Configurable: identifies a design element that can be configured or altered to produce a different Configuration (C.20) or behaviour. Such change is generally performed by the user or buyer of the software, not the by vendor or its development team</p> <p><i>kind:</i> the kind of this configurable design element. Examples: “Memory Bits”, “Lookup Tables”, ...etc</p> <p><i>when:</i> specifies when this configurable design element can be configured to change configurations. Examples: “Compile-Time”, “Link-Time”, “Run-Time”, ... etc</p>
<p>(C.22) Loadable: identifies a design element that can be loaded by the user to change the Configuration (C.20). Loadable design elements are loaded on Configurable (C.21) design elements</p>
<p>(C.23) Configurator: identifies a design element that can configure Configurable (C.21) design elements to change the Configuration (C.20), possibly by loading Loadable (C.22) design elements</p>
<p>(C.24) Replicated: identifies a design element that participates in a Replication Group (C.26), such as multiple-version dissimilar software, and whose output is evaluated by a Comparator (C.25)</p> <p><i>id:</i> specifies a unique identifier for this entity within its replication group. Examples: “Filter Version 1”, “Filter Version 2”, “Filter Version 3”, ... etc</p>
<p>(C.25) Comparator: identifies a design element that analyzes outputs of Replicated (C.24) design elements and determines the formal output of the Replication Group (C.26)</p> <p><i>policy parameters:</i> specifies how the comparator determines the formal output. Can include assignment of weights. Examples: “Equal Weights”, “Majority Voting”, ... etc</p>
<p>(C.26) ReplicationGroup: identifies a software replication group composed of Replicated (C.24) design elements and a Comparator (C.25) that compares their outputs. For example, a replication group is an instance of software redundancy or multiple-version dissimilar software. It is a technical solution to reliability challenges and has been traditionally used in safety-critical systems</p> <p><i>id:</i> specifies the ID of this replication group. Examples: “Radar Filter Replication Group”, “Controller Replication Group”, “REPLICATION 1”, ... etc</p>

Table 11: Relationships between safety-related concepts

<p>Requirement [0..*] Is Requirement Of Requirement [0..*] Each Requirement may be traceable to zero or more higher-level Requirements. Conversely, a Requirement may have zero or more lower-level Requirements (traceable to it)</p>
<p>Deviation [0..*] References Requirement [1..*] Each Deviation must deviate from at least one, potentially more, Requirement. Moreover, there may exist more than one Deviation from a particular Requirement. However, not every Requirement may have Deviations from it, which would be the case when the design fully conforms to the Requirements</p>
<p>ImplementationStyle Is Child Class Of Style Each ImplementationStyle is a Style</p>
<p>ImplementationStyle [0..*] References Requirement [0..*] Each ImplementationStyle may conform to, or deviate from, zero or more Requirements. Conversely, a Requirement may require zero or more ImplementationStyles. In the case where an ImplementationStyle is not associated with any Requirements, the ImplementationStyle signifies a design decision rather than an obligation or a requirement</p>
<p>BehaviouralStyle Is Child Class Of Style Each BehaviouralStyle is a Style</p>
<p>BehaviouralStyle [0..*] References Requirement [0..*] Each BehaviouralStyle may conform to, or deviate from, zero or more Requirements. Conversely, a Requirement may require zero or more BehaviouralStyles. In the case where a BehaviouralStyle is not associated with any Requirements, the BehaviouralStyle signifies a design decision rather than an obligation or a requirement</p>
<p>Nature [0..*] References Requirement [0..*] A Nature may have been used solely as a design decision, in which case it is not associated with any Requirements, or it may have been used to conform to one or more Requirements. Conversely, a Requirement may exist but not cause any Natures, or it may cause one or more Natures</p>

<p><i>Rationale</i> [0..*] References Requirement [1..*] Each Rationale must be associated with at least one, potentially more, Requirement. Moreover, there may exist more than one Rationale associated with a particular Requirement. However, not every Requirement may have Rationales associated with it. However, such a case is uncommon because it would mean that there are no design elements traceable to this Requirement</p>
<p><i>Reaction</i> Is Child Of Event Each Reaction is an Event</p>
<p><i>Reaction</i> [0..*] Is Consequence Of Event [1..*] Each Reaction is a consequence of one or more Events because it is executed in response to the Events. However, each Event may not cause any reactions at all, or it may cause one or several Reactions. Since Reactions are Events by inheritance, then a terminal Reaction, which is the last Reaction in a chain of Reactions, does not cause any more Reactions</p>
<p><i>SafetyCritical</i> [1..*] Triggers Event [0..*] A SafetyCritical entity may trigger zero or or more Events. A particular Event may not be triggered by any SafetyCritical entity, or it may be triggered by one or more SafetyCritical entities</p>
<p><i>Partition</i> [0..*] References Requirement [0..*] A Partition may exist to fulfill one or more Requirements, or it may exist as a design decision to isolate functionally independent elements such that a failure in one component does not cause the other to fail. Conversely, a Requirement may or may not require one or more Partitions to be performed</p>
<p><i>Partition</i> [0..*] Is Partitioned From SafetyCritical [1..*] By definition, a Partition is always Partitioned from one or more SafetyCritical entities. However, a SafetyCritical entity may not necessarily have one or more Partitions from it</p>
<p><i>Handler</i> [0..*] Handles Event [1..*] A Handler handles at least one Event, and it usually handles more than one Event. However, one or more Events may not necessarily be handled by a Handler. The latter case may occur for Events that are not of interest in the system, such as non-safety-critical events. In addition, it usually occurs for many Reactions, which are Events by inheritance</p>
<p><i>Handler</i> [0..*] Performs Reaction [1..*] A Handler performs one or more Reactions. However, a Reaction may not necessarily be performed by a Handler, or it may be performed by one or more Handlers</p>
<p><i>Monitor</i> [0..*] Monitors SafetyCritical [1..*] A Monitor monitors one or more SafetyCritical entities. However, not every SafetyCritical entity is monitored by a monitor. It is also possible for a SafetyCritical entity to be monitored by more than one Monitor</p>
<p><i>Monitor</i> [0..*] Detects Event [1..*] A Monitor detects at least, but usually more than, one Event. However, an Event may go undetected by Monitors, or it may be detected by one or more Monitors</p>
<p><i>Monitor</i> [1..*] Notifies Handler [0..*] Each Handler is notified by at least one Monitor. However, some Monitors may not necessarily notify any Handlers, and a Monitor may notify more than one Handler</p>
<p><i>EnvironmentModel</i> [0..*] Models SafetyCritical [1..*] An EnvironmentModel models at least one SafetyCritical entity. A SafetyCritical entity may not have any EnvironmentModels, or it may have one or more EnvironmentModels. For example, a radar system may have two EnvironmentModels, with each one modeling the radar's behaviour under different environmental conditions.</p>
<p><i>Strategy</i> [0..1] Describes Strategy Of SafetyCritical [1..*] A Strategy describes the approach used (e.g. design strategy, testing strategy) in one or more SafetyCritical entities. In addition, a SafetyCritical entity's approach may, or may not, be described by a Strategy</p>
<p><i>Measure</i> [0..1] Quantifies Characteristic Of SafetyCritical [1..*] A Measure quantifies a characteristic of one or more SafetyCritical entities. In addition, a SafetyCritical entity may, or may not, have characteristics quantified by a Measure.</p>
<p><i>Interface</i> [0..*] Is Interface For SafetyCritical [1..*] Each Interface is for one or more SafetyCritical entities or components. In addition, a specific SafetyEntity may have one or more Interfaces. An example of the latter case would be where a subsystem has one Interface to it in each of the other subsystems in the complete system</p>

<p><i>Concurrent [0..*] Triggers Event [0..*]</i> Each Concurrent entity may trigger zero or more Events. Conversely, each Event may be triggered by zero or more Concurrent entities. A Concurrent entity may not trigger any Events if it is passive</p>
<p><i>Defensive [0..1] Performs Reaction [1..*]</i> A Defensive entity protects against unusual inputs by performing one or more Reactions to such unusual inputs, or Events. However, Reactions are not necessarily performed by Defensive entities</p>
<p><i>Configurable [0..*] Defaults To Configuration [1..1]</i> Each Configurable entity must be defaulted to a particular Configuration. In addition, a Configuration may be the default one for zero or more Configurables</p>
<p><i>Configurable [1..*] Is Configurable To Configuration [1..*]</i> Each Configurable entity may be configured to produce one or more Configurations. In addition, each Configuration can be produces by configuration one or more Configurable entities in a particular way</p>
<p><i>Loadable [1..*] Is Loadable On Configurable [1..*]</i> Each Loadable entity is loadable on one or more Configurable entities. Conversely, every Configurable entity can be configured by loading one or more Loadables on it</p>
<p><i>Loadable [0..*] Requires Configuration [0..*]</i> Loading a Loadable entity may require specific base Configurations for it to be Loaded. For example, loading a particular software patch may require pre-loading earlier patches. However, there may not be such a requirement if the patch is a complete and comprehensive patch, rather than an incremental patch. Conversely, not every Configuration is required by Loadable entities</p>
<p><i>Loadable [1..*] Produces Configuration [1..*]</i> A Configuration may be produced by loading a Loadable. A Loadable may produce more than Configuration if loaded on different base Configurations. For a Configuration to be produced, at least one Loadable must be loaded</p>
<p><i>Configurator [1..*] Configures Configurable [1..*]</i> A Configurator configures one or more Configurable entities. A Configurable entity may be configured by more than one Configurator, such as the case where the Configurators configure different aspects of the Configurable entity</p>
<p><i>Configurator [1..*] Loads Loadable [1..*]</i> A Configurator loads one or more Loadables. In addition, a Loadable is loaded by one or more Configurators</p>
<p><i>Comparator [1..1] Compares Replicated [2..*]</i> A Comparator compares the outputs of at least two Replicated entities. The output of a Replicated entity is compared by exactly one Comparator</p>
<p><i>ReplicationGroup [1..1] Owns Comparator [1..1]</i> Each ReplicationGroup has exactly one Comparator</p>
<p><i>ReplicationGroup [1..1] Owns Replicated [2..*]</i> Each ReplicationGroup has at least two Replicated entities</p>