An Assurance Level Sensitive UML Profile for Supporting DO-178C

by

Nicolas Metayer

THESIS PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
IN PARTIAL FULFILLMENT OF A MASTER'S DEGREE
WITH THESIS IN SOFTWARE ENGINEERING
M.A.Sc.

MONTREAL, "FEBRUARY 1, 2018"

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

BOARD OF EXAMINERS

THIS THESIS HAS BEEN EVALUATED

BY THE FOLLOWING BOARD OF EXAMINERS

Mrs. Ghizlane El Boussaidi, Memorandum Supervisor
Department of Software and IT Engineering, École de technologie supérieure

M. Segla Kpodjedo, President of the Board of Examiners
Department of Software and IT Engineering, École de technologie supérieure

M. Abdelouahed Gherbi, Member of the jury
Department of Software and IT Engineering, École de technologie supérieure

THIS THESIS WAS PRESENTED AND DEFENDED

IN THE PRESENCE OF A BOARD OF EXAMINERS AND THE PUBLIC

ON "JANUARY 26, 2018"

AT ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Mrs Ghizlane El Boussaidi, for the opportunity to work on this research project and for the quality and quantity of the guidance provided throughout the fulfillment of my master. I would also like to thank my colleagues from the LASI laboratory for their support and advices. I would especially like to thank Christel for the welcoming support he has provided for my integration in Montreal and Andrès for all of the feedback he offered while we worked together on this research project.

I would like to thank the members of the jury that accepted to review this thesis and for the provided feedback.

I would like to thank Mrs Duvalet, who supported my application to pursue a joint degree between ETS and 3IL. I will always be grateful for letting me have this great opportunity to pursue a master degree at Ecole de Technologie Superieure.

I would like to thank my family, especially my mother and my sister for their unconditional support and encouragement that greatly helped me during my researches. Also I would like to thank my dear friends Thibaut, Anatole, Matthieu and Valentin for their support during the fulfillment of my master.

UN PROFIL UML MODULAIRE SELON LE NIVEAU D'ASSURANCE LOGICIELLE POUR LE SUPPORT DE DO-178C

Nicolas Metayer

RÉSUMÉ

Plusieurs approches basées sur les modèles ont été proposées pour faciliter le développement de logiciel critique certifiable. Dans ce mémoire, nous nous intéressons aux logiciels avioniques devant se conformer au standard DO-178C. Cependant, les approches existantes ne supportent pas entièrement les activités du cycle de vie logiciel défini par DO-178C.

Dans ce mémoire, nous proposons un profil UML capturant les concepts de DO-178C ainsi que ses suppléments afin de modéliser les évidences exigés pour la certification. Ce profile fournit des éléments de modélisation pour définir un cycle de vie logiciel conforme à DO-178C, pour spécifier des exigences logicielles et des données de vérification, ainsi que pour spécifier la traçabilité demandée par DO-178C. De plus, ce profil à l'unique caractéristique de fournir un moyen de spécifier les objectifs et activités à effectuer durant le cycle de vie logiciel selon le niveau d'assurance logiciel visée ainsi que des suppléments à DO-178C utilisés.

Nous avons implémenté the profil proposé au sein de Papyrus, un environnement de modélisation pour UML. Nous avons utilisé le profil pour modéliser un exemple réaliste de logiciel avionique. En particulier, nous avons illustré l'utilisation de ce profile à travers quatre cas d'utilisation.

Mots-clés: Logiciel avionique, certification logiciel, DO-178C, Ingénierie dirigée par les modèles, Langage de modélisation dédié, Profil UML

AN ASSURANCE LEVEL SENSITIVE UML PROFILE FOR SUPPORTING DO-178C

Nicolas Metayer

ABSTRACT

Several model-based approaches have been proposed to ease the process of developing certifiable safety-critical software. In this thesis, we are interested in airborne software which must comply with DO-178C standard. However, existing approaches do not provide complete support for all the activities of the software life cycle as defined by DO-178C.

In this thesis, we propose an UML profile that captures the concepts of DO-178C and its supplements in order to model the evidence required for certification. This profile provides modeling constructs for the definition of a DO-178C compliant software life cycle, the specification of the software requirements, the specification of verification data and finally the specification of the traceability that is requested by DO-178C. Furthermore, this profile has the unique feature of providing means to specify the objectives and activities to be performed throughout the software life cycle depending on the targeted assurance level and applied DO-178C supplements.

We implemented the proposed profile within Papyrus, an UML modeling environment. We used the profile to model a realistic example of airborne software. Specifically, we illustrated the usefulness of the profile through four use cases.

Keywords: Airborne software, software certification, DO-178C, model-driven engineering, domain specific modeling language, UML profile

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Page

LIST OF ABREVIATIONS

DSML	Domain Specific Modeling Language
FAA	Federal Aviation Administration
HLR	High-Level Requirement
LLR	Low-Level Requirement
MDE	Model Driven Engineering
SRATS	System Requirement Allocated to Software
UML	Unified Modeling Language

INTRODUCTION

Research context

Safety-critical systems are more and more relying on software. Among these systems, avionics are increasingly depending on software to control their behaviors (Huhn & Hungar, 2007; Pettit *et al.*, 2014). Because failure in aircraft systems could result in multiple fatalities, a high-level of confidence in the ability to operate safely an aircraft is required (Marques *et al.*, 2012; Gallina & Andrews, 2016). As such, safety is one of the major concerns in the avionic domain. Although safety is considered to be a problem related to physical systems, software can contribute to hazards (Heimdahl, 2007; Rushby, 2007). Such hazards are the result of erroneous control of the system by the software. Demonstrating that an airborne software complies with its assigned level of safety is not a trivial process.

To ensure that the necessary safety evidence, defined by Nair *et al.* (2014) as «artifacts that contribute to developing confidence in the safe operation of a system» are provided, the activities related to the development of airborne software are strongly regulated (Nejati *et al.*, 2014). Such regulation exists to guide and, in some cases, to enforce certain practices related to software engineering in order to gain the required confidence in the safety of the produced software. Regulations that apply to airborne software development include DO-178, Software Considerations in Airborne Systems and Equipment Certification.

Initially released in 1982, DO-178 has been developed to provide the industry with guidelines for developing airborne software to satisfy the airworthiness requirements. Guidelines offered by DO-178 do not prescribe how the software development should be performed (Rushby, 2007). However the guidelines specify which activities should be performed and documents should be produced. As for many domains, software engineering methodologies are evolving. To keep pace with such changes and allow the industry to benefit from these new methodologies, the regulation corpus evolves. Although happening at a slow rate, multiple revisions of DO-178

were developed since its initial release. DO-178A in 1985, DO-178B in 1992 and finally DO-178C in 2011 (RTCA, 2011). This latest revision removes some ambiguities found in DO-178B guidelines and provides additional guidelines in the form of supplements addressing new software development technologies (i.e. object-oriented programming, formal methods). These supplements are DO-331 (RTCA, 2011c) for model-based development and verification, DO-332 (RTCA, 2011d) for object-oriented technologies and DO-333 (RTCA, 2011e) for the use of formal methods. Furthermore, tool qualification guidelines are provided by DO-330 (RTCA, 2011b). As the release of DO-178C is still fairly recent, the industry still needs to adapt their development practices to benefit from the use of the technologies that are addressed by the supplements.

Among the recent development made in software engineering technologies, industrials are particularly interested in approaches that use model-driven engineering (MDE) methodology. MDE is a software engineering approach that aims at alleviating the complexity of software development through the use of domain specific models and transformations that support the refinement of these models into artifacts (Schmidt, 2006).

Research problem statement

Development activities of airborne software are guided by DO-178C and its supplements. The scope of the guidelines covers the complete software development life cycle. The software life cycle, as defined by DO-178C, is comprised of a set of processes that are determined by an organization to be sufficient and adequate for the development of the software product according to the criticality of the software. The software life cycle begins when the decision to produce the software is made and ends when the product is retired from service.

Prior to actually develop the software, planning of activities constituting the software life cycle shall be performed. Thus, in the context of DO-178C, the planning process is the first process

to be carried out. The planning process is an important process as it defines the activities to be performed to produce the software and it describes all the data that will be produced; these data will serve as evidence for certification. The planning process produces a number of plans including the Plan for Software Aspects of Certification (PSAC) which has to be submitted first to the certifying authorities. Only on approval of the PSAC can the software development activities begin. Hence, the PSAC must demonstrate that the proposed software life cycle is compliant with DO-178C. However most existing model-driven approaches that support the development of airborne software according to DO-178C do not offer any support to automatically create the PSAC and all the plans that need to be produced during the planning process depending on the criticality level of the software.

Once the PSAC is approved by the certifying authorities, the actual process of developing the software can begin. Both the software development and verification processes can start. Software development processes include requirements specification, design, coding and integration activities. Specifying requirement involves the development of high-level requirements (HLRs) from the system requirements allocated to software (SRATS) while design involves the development of the low-level requirements (LLRs) and the software architecture from the high-level requirements. The software verification process consists in carrying out a number of tests, reviews and analyses. Thus data produced by this process include test cases, test procedures, test execution results, reviews and analyses specifications, and reviews and analyses results. To comply with DO-178C, we need to explicitly establish traceability between the requirements and the verification data. Moreover, we need to trace both LLRs and software architecture to HLRs. The traceability must be established both forward and backward. Flaws in the traceability of the artifacts produced during the software life cycle result in major non-compliance issues (Nejati *et al.* 2012) that impede the certification of the software.

Existing model-driven approaches that support DO-178C focused on the software requirements process and software design process (Zoughbi *et al.*, 2010; WU *et al.*, 2015) and the software verification process (Stallbaum & Rzepka, 2010). These approaches do not model the DO-178 standard. The RAF meta-model (De la Vara *et al.*, 2016) is an exception as it is built from a number of standards, including DO-178C. However it uses a unified vocabulary that is not specific to DO-178C. Moreover, the existing approaches do not tackle the assurance level modularity introduced by DO-178. As a result these approaches do not consider the variations in the compliance needs to be provided in order to achieve certification.

Thesis organization

The reminder of this thesis is organized as follows. In chapter 1, we provide an introduction to DO-178C and we discuss the existing works related to the development of safety-critical systems using a model-based engineering methodology. Chapter 2 introduces our research objectives and describes the proposed approach along with the methodology that guided our work. Chapter 3 provides the description of our conceptual model of DO-178C. Chapter 4 describes the proposed UML profile for DO-178C. Chapter 5 describes the integration of our profile within an open-source UML modeling tool and demonstrates the use of the proposed profile through 4 use cases by modeling a landing gear control software. Finally, we conclude our thesis, introduce the limitations we faced and provide possible future work to extend our approach.

CHAPTER 1

LITERATURE REVIEW

In this chapter, we introduce key concepts related to this thesis and discuss relevant existing approaches. In Section 1.1, we first give an overview of DO-178C and its needs toward achieving software certification. We also provide in Section 1.2, an overview of the MDE methodology and domain-specific modeling. In particular, we describe the unified modeling language and its profile extension mechanism. In section 1.3, we present relevant existing approaches for modeling safety critical systems. Finally, in Section 1.4 we discuss the findings of the previous sections and highlight the limitations of the existing work.

1.1 DO-178C

DO-178, "Software Considerations in Airborne Systems and Equipment Certification" (RTCA, 2011a) is the de-facto safety standard used to drive the development activities of airborne software systems. Its purpose is to provide guidance for the development of software products in respect of the airworthiness requirements assigned to the software. The rigor of the airworthiness requirements assigned to a software product is dependent on the product's associated criticality. To address the different levels of criticality defined at the system level, the guidelines prescribed by DO-178 are organized in a modular manner. To provide such modularity, the standard defines five software levels, often referred to as design assurance level (DAL) or assurance level (Rushby, 2011). These assurance levels are mapped to the following failure condition categories that are defined in the Federal Aviation Administration's Advisory Circular (AC) 25.1309 (FAA, 1988):

- **Catastrophic**: Defines failure conditions that would usually lead to the loss of the aircraft, thus leading to multiple casualties.
- **Hazardous**: Defines failure conditions that lead to the reduction of the ability to operate the aircraft within acceptable safety margins. Such failure conditions could result in 1)

an important decrease in the functional capabilities of the airplane, 2) physical agony or increase of the aircraft operating crew workload, resulting in the loss of the ability for the crew to perform their tasks as intended or to perform them at all, or 3) severe injuries or fatalities inflicted to a small number of passengers other than the flight crew.

- **Major**: Defines failure conditions that lead to a reduction of ability of the aircraft or its operating crew to deal with unexpected operating conditions. Such failure conditions would result in 1) a reduction of the functional capabilities or safety margins of the aircraft, 2) an increased workload for the crew or conditions that hinder its efficiency, or 3) physical distress to passengers or crew members with possible injuries.
- **Minor**: Defines failure conditions that would not lead to an important reduction of the aircraft safety. However such conditions involve a small increase of the crew workload that remains well within its capabilities.
- **No safety effect**: Defines failure conditions that do not have an impact on safety. Such failures do not impact the operational capabilities of the aircraft nor the aircraft operating crew workload.

DO-178 defines software levels that map to the above described failure conditions as follows:

- **Level A** : Defines software whose undesired behavior, as outlined by the safety assessment process, would be contributing to or resulting in a system malfunction whose consequence would result in a catastrophic failure condition for the aircraft.
- **Level B**: Defines software whose undesired behavior, as outlined by the safety assessment process, would be contributing to or causing a system malfunction whose consequence would result in an hazardous failure condition for the aircraft.
- **Level C**: Defines software whose undesired behavior, as outlined by the safety assessment process, would be contributing to or causing a system malfunction whose consequence would result in a major failure condition for the aircraft.

- **Level D**: Defines software whose undesired behavior, as outlined by the safety assessment process, would be contributing to or causing a system malfunction whose consequence would result in a minor failure condition for the aircraft.
- **Level E**: Defines software whose undesired behavior, as outlined by the safety assessment process, would be contributing to or causing a system malfunction that would have no impact on the aircraft.

DO-178 software levels are used to guide an applicant, i.e. an organization that applies for the certification of its software product, in the definition of the software life cycle to be applied to develop the software product. Applicants are guided in terms of activities to be performed and objectives to be achieved during the software life cycle. The number of activities and objectives is dependent on the software level assigned to a software product. In its latest revision, DO-178C which has been published in 2011 to address identified issues in the text of its predecessor (DO-178B), the number of objectives to be achieved ranges from 26 for software level D to 71 for software level A.

Applicants are not forced to follow DO-178C to show compliance with the applicable airworthiness regulation affecting software in airborne systems (FAA, 2013). Other means of compliance can be used by an applicant if an appropriate level of assurance can be demonstrated for the software product. However it is strongly recommended to follow the workflow prescribed by DO-178C in order to demonstrate that a product complies with its assigned airworthiness requirements. In fact DO-178C prescribes a software life cycle that is decomposed into a set of processes as depicted in Figure 1.1. These processes are the following:

- The *software planning process* in charge of defining and coordinating the activities driving the development and integral processes.
- The *software development processes* in charge of the production of the software product. Such processes include the software requirements process, the software design process, the software coding process and the integration process.

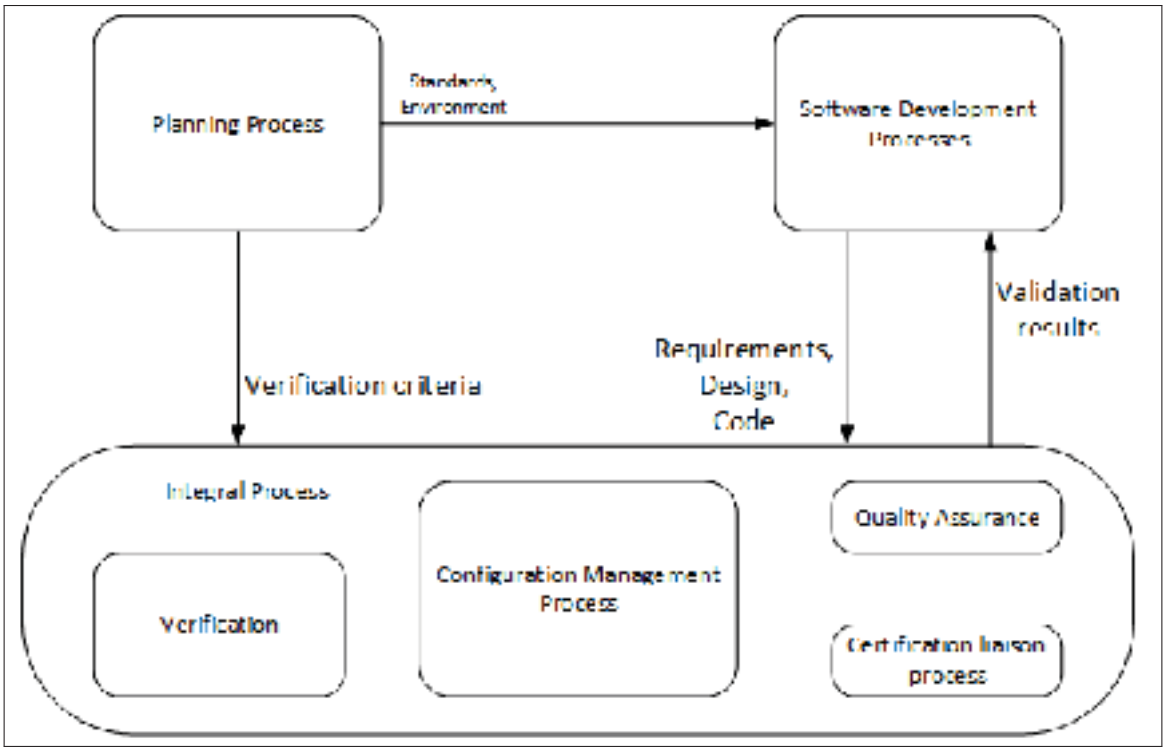


Figure 1.1 DO-178 processes.

- The *integral processes* in charge of ensuring the correctness and control of, and confidence in the software life cycle processes and their outputs. Such processes include the software verification process, the software configuration management process, the software quality assurance process, and the certification liaison process. These processes should be performed in concurrence with the planning and development processes throughout the software life cycle.

Figure 1.2 offers an overview of the workflow to be conducted during the software development processes and the software verification process. The development of DO-178C compliant software begins with the definition of the high-level requirements (HLRs), obtained by refinement of the system requirements allocated to software [1]. Review and analysis of the HLRs are then conducted to assess the various properties they must exhibit such as HLRs consistency and

[1] The definition of the system requirements allocated to software is out of the scope of DO-178 guidelines.

compliance with the system requirements allocated to software, their capabilities to be verified, and that appropriate justification is provided when necessary. With the HLRs validated, the software design process can be initiated leading to the development of the low-level requirements (LLRs) and the software architecture by refinement of the HLRs. The LLRs are assessed by combination of review, analysis and test cases to ensure that the developed product complies with its high-level requirements.

In DO-178C, the current version of the standard, few modifications have been made to the core document. Theses modifications were developed with the objective of maintaining backward compatibility with DO-178B and they mainly aim at fixing errors, inconsistencies and using a more consistent terminology. The most notable change lies in the introduction of supplements to address issues related to the use of new software development technologies that were not addressed at the time of DO-178B release. The supplementary documents cannot be used as standalone documents, therefore they must be used as additional guidelines to DO-178C. These supplements include tool qualification guidelines, model based development and verification, object oriented technologies, and formal methods. These supplements may add, delete or modify objectives, activities and life cycle data defined in DO-178C. As such, compliance with the corresponding supplement(s) is required when one of the addressed technology is used. When using supplements, a project's plan for software aspects of certification should specify which supplements are in use and how they are intended to be used (FAA, 2013). In the following subsections, we briefly introduce each supplement and its major concepts.

1.1.1 DO-331

DO-331, model-based development and verification supplement to DO-178C and DO-278A, introduces guidelines pertaining to the use of model based development technologies to perform the software development activities. A model is defined by DO-331 (RTCA, 2011c) as an abstract representation of a system aspect used to perform analysis, verification, simulation, code generation or a combination of these activities. Models have to be unambiguous regardless of the level of abstraction used to capture the system in order to enable the aforementioned

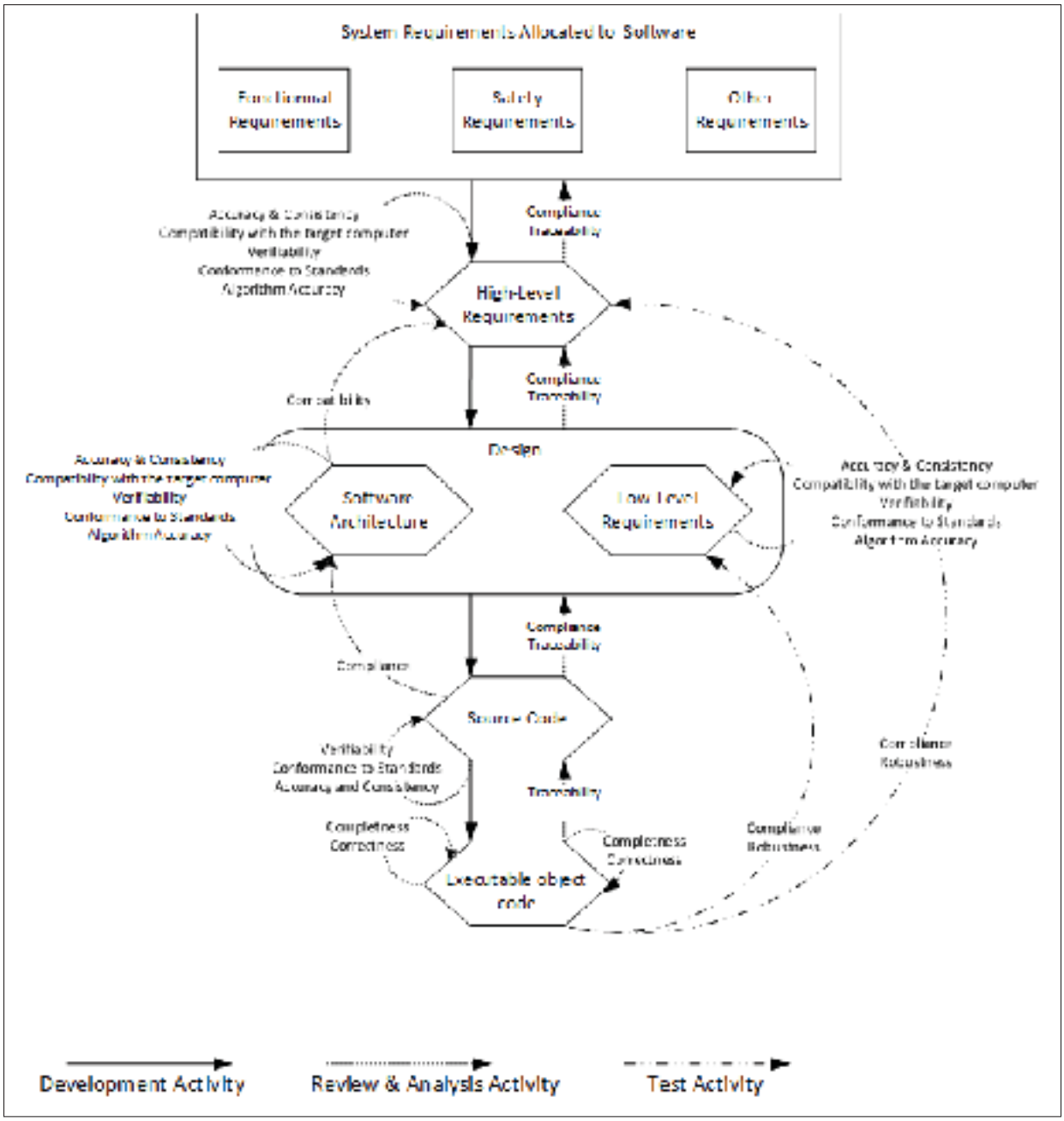


Figure 1.2 DO-178C software development and verification workflow. Adapted from (RTCA, 2011e).

activities. DO-331 distinguishes two types of models: specification models and design models. In the context of DO-331, a model cannot be classified as both specification and design. Examples of model usage scenarios within the scope of DO-178 are provided in Table 1.1.

Table 1.1 Examples of model usage. Adapted from (RTCA, 2011c).

Process generating life cycle data	**Example 1**	**Example 2**	**Example 3**	**Example 4**
Software Requirements and Software Design Processes	Requirements from which the model is developed	Specification model	Specification model	Design model
	Design model	Design model	Textual description	
Software Coding Process	Source Code	Source Code	Source Code	Source Code

Specification models capture high-level requirements to provide an abstract representation of the functional, performance, interface, or safety properties of a software component. Specification models do not capture details of the software such as internal data structures, external data flow, or internal control flow. Design models capture the software low-level requirements and/or the software architecture. Design models might capture algorithms, software components internal data structures, and data and control flow. They may be used to generate the source code.

The use of DO-331 does not relieve an applicant from performing the objectives of DO-178C. As models represent either HLRs, LLrs and/or the software architecture, models have to be treated in the same manner as the artifact they represent, meaning that traceability as defined in DO-178C has to be maintained when using models. Such traceability in a model based development environment includes the traces between the source code and design models, the traces between the design elements and their related specification models, and the traces between specification models and the system requirements allocated to software.

1.1.2 DO-332

Object oriented programming paradigm has been developed in the 1950s. Although being widely used for non-critical software development, its use for safety critical application for

avionics software only increased recently. To address issues introduced by object oriented technology and related techniques, the DO-332 supplement was released to provide guidance when using such programming paradigm.

An object oriented technology is a software development methodology where the software design is expressed using objects and their interrelationships. The technology also makes use of techniques such as inheritance, polymorphism, overloading, type conversion, exception, dynamic memory management and virtualization. Because object oriented technology vastly differs from the traditional approach of procedural programing, it raises specific issues when used in the design and implementation of software for airborne systems. These issues are addressed by DO-332 (RTCA, 2011d).

For the software development activities for instance, DO-332 introduces compliance needs regarding the definition of a class hierarchy deriving from the high-level requirements, the definition of local type consistency where substitution is used, the definition of strategies related to dynamic memory management and exception management.

In order to enable the verification of an object oriented design, traceability as defined by DO-178C has to be maintained. In the context of DO-332, this traceability includes the development of bi-directional traces between the requirements and the methods (of the classes) that implement these requirements. Verification activities for software developed following an object oriented technology have to comply with the verification objectives of DO-178C. DO-332 introduces further objectives with focus on the verification of the class hierarchy for consistency with the high-level requirements, local type consistency wherever inheritance method overriding and dynamic dispatch are used. Finally the guidelines provide emphasis on the verification of the correct implementation of the dynamic memory management strategies.

1.1.3 DO-333

Formal methods are mathematical based techniques used to specify, develop and verify aspects of software (RTCA, 2011e). Such methods have been considered for avionic software before the

release of DO-178B. However formal methods were not widely used at the time and DO-178B did not provide clear guidance for their use. To provide precise guidance on the matter, DO-333 Formal Methods Supplement to DO-178C and DO-278A introduces guidelines for applicants using formal methods as a mean to achieve the development and verification objectives of DO-178C.

In the context of DO-333, a formal method is defined as the combination of a formal model and a formal analysis. A formal model, as defined by DO-333, is an abstract representation of a given aspect of a system, however such a model is defined using a formal notation having a precise and unambiguous, mathematically defined syntax and semantic. A formal analysis is the application of mathematical reasoning about a formal model to guarantee that properties, defined by the software requirements, are always satisfied. Formal analyses enable the automation and exhaustive verification of model properties. They are classified by DO-333 into three categories: 1) deductive method, 2) model checking and 3) abstract interpretation.

1.2 Model-driven engineering

Model-driven engineering (MDE) is a software engineering approach that aims at alleviating the complexity of software development through the use of domain specific models and transformations that support the refinement of these models into artifacts (Schmidt, 2006). Thus an MDE approach uses models as the main artifact of the software life cycle. Models are specified using modeling languages. Modeling languages are defined with a combination of the following elements (Atkinson & Kuhne, 2003): I) a concrete syntax or the notation used to build the models; II) an abstract syntax or vocabulary of concepts that are part of the language; III) a semantic, either implicit or explicit, defining the well formedness rules of the language; and IV) a mapping between the abstract and the concrete syntax. These properties are specified by the language meta-model.

A meta-model is a model that represents the concepts, associations, and constraints that form the definition of a language (Atkinson, 2003). Models created using a modeling language

are said to be in «conformance» with the modeling language's meta-model which is in turn in conformance with its own meta-model. Because meta-models are also models, they are represented using a modeling language called meta-meta-model. However this way to define meta-models introduces, in theory, an infinite number of meta-meta-model definitions. To cope with this problem, the Object Management Group (OMG) introduced a meta-modeling language within its four-level modeling framework, the Meta-Object Facility (MOF) (OMG, 2015). MOF is a meta modeling language for defining other modeling languages, including MOF itself.

Figure 1.3 provides an overview of the four-level modeling framework introduced by the OMG. Models at the M0 level represent entities of the real world that are to be modeled (for example an aircraft). Models at the M1 level are the actual models (e.g. a UML state machine diagram) that are created using the semantics and notations defined at the meta-model (M2) level. The Unified Modeling language (UML) is one of the most common modeling language of level M2 whose base meta-meta-model is MOF (M3).

Models may be used by various stakeholders to represent different concerns of a system. In this context, models are used with two distinct objectives to reason about the system (Selic, 2007): 1) provide multiple perspectives on the system and 2) provide multiple levels of abstraction. The former, commonly referred to as «views» are the representation of various concerns of the system. A single view describes in details a specific concern of a system and when grouped together, views provide the complete description of the modeled system. Regarding the abstraction levels, models at the highest level are closer to the domain's problem and those at the lowest level include implementation details. Huhn & Hungar (2010) identified use cases where platform independent models can be used when developing software related to safety-critical systems. Models can be used at every stage of the software life cycle. These use cases are: (1) the specification of the software requirements, (2) the definition of the software architecture and its evolution, (3) code generation, (4) verification,(5) validation and (6) certification.

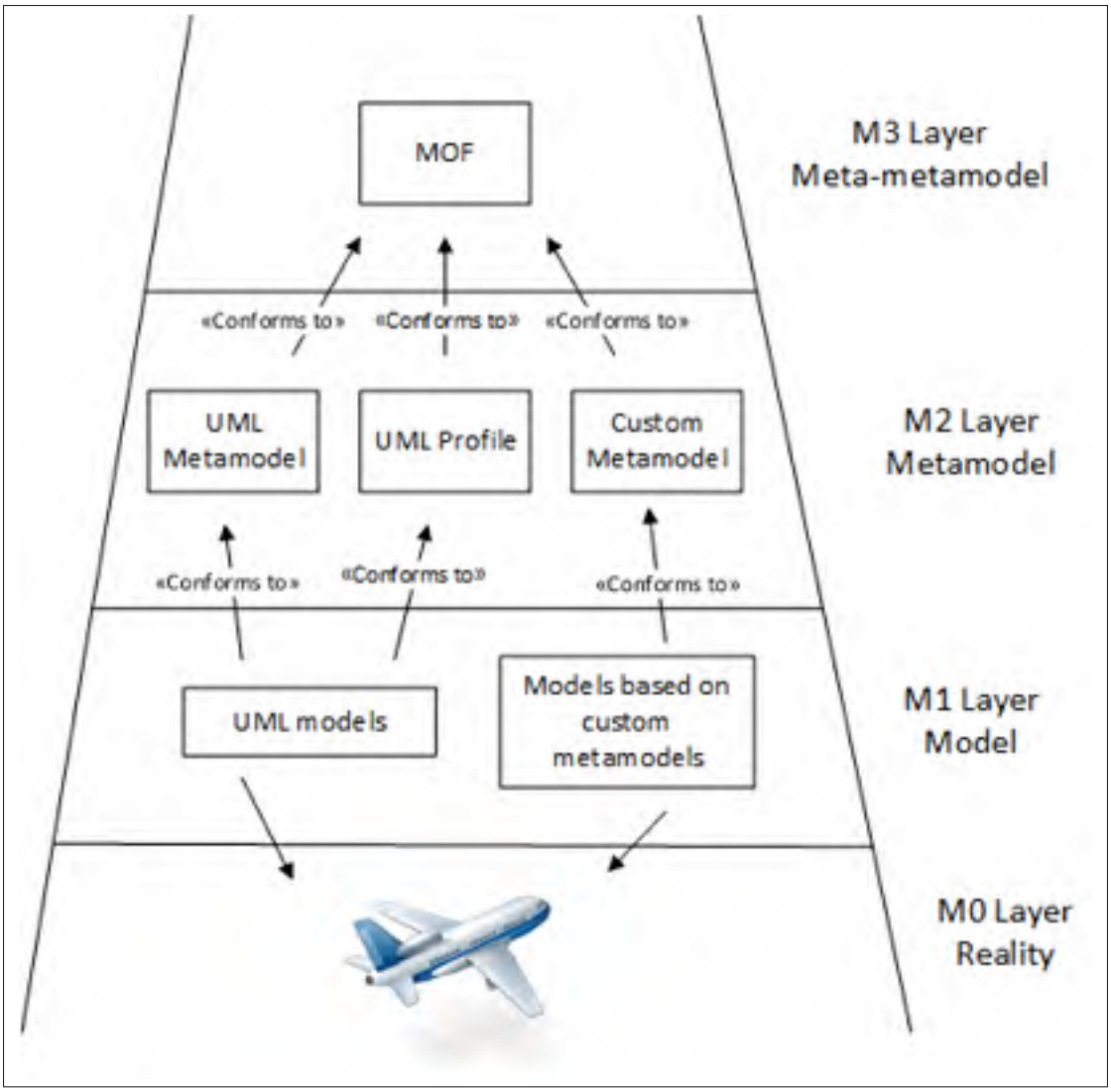


Figure 1.3 OMG's four-level architecture. Adapted from Djurić *et al.* (2005)

1.2.1 Domain specific modeling

To face problems emerging in specific domains of application, domain specific languages have proven their efficiency to overcome the complexity of software development project (Voelter *et al.*, 2013). In the context of model driven engineering, there is an increase in the use of domain specific modeling languages (DSML) because they enable (Voelter *et al.*, 2013): 1) a better expression of the solutions to the problems faced in a particular domain of application by using dialect and constructs pertaining to the domain, and 2) the capture of the domain knowledge,

easing its exchange and reuse among the involved stakeholders. Examples of DSML include Simulink, SCADE, MARTE or SysMl.

According to Selic (2007) and Lagarde *et al.* (2008), there exist three primary methods for the creation of a domain specific modeling language, two of which are based on an existing language: 1) the extension of an existing modeling language, 2) the refinement of an existing language and 3) the definition of the modeling language from scratch.

Among these methods, the refinement of an existing modeling languages is the most practical and cost effective solution to design a domain specific modeling languages (Selic, 2007). The reason lies behind the quantity of reuse that such solution allows. Indeed, existing language might provide an extension mechanism (i.e. UML profile) and tools might provide support for such mechanism. Finally, the refinement of an existing language requires less training to become familiar with the refined language.

The extension or refinement of an existing modeling language are the preferred methods among modeling language designers due to the benefit offered by these two methods. Using an existing modeling language as a basis allows designers to benefit from the knowledge revolving around the used technology to better tackle a domain's problem. Furthermore these two methods allow a faster integration of new domain specific modeling languages within the development teams. One such modeling language that allows its extension and refinement is the unified modeling language (UML). Extension is done through mechanisms provided by the language while refinement add new concepts to the language which might introduce some incompatibilities with existing tool and environments.

1.2.2 UML and its extension mechanisms

The unified modeling language (UML) is an OMG's standardized general purpose modeling language. It is a de-facto modeling language used throughout the software development life cycle: specification, design, and documentation. UML is used in a broad range of areas such as system, hardware, and even business process modeling. This wide usage is due to two

reasons. The first one is because UML is a general purpose language that enables to represent a system using multiple views and with different levels of abstraction. The second reason is the UML capability to be both extended and/or refined for the specific needs of a domain. The UML meta-model provides a built-in mechanism, called UML profiles to support the extension approach to designing domain specific modeling language.

UML profiles have the advantage, compared to the refinement of an existing meta-model, of reducing the cost to develop a domain specific modeling language. In fact, a number of existing UML modeling tools support the definition of UML profiles. Furthermore, the cost of training people to use UML profiles is greatly reduced because software engineers are generally familiar with UML and its profile mechanism. The effort required to define the syntax and semantic of a domain specific modeling language using the UML profile mechanism is also reduced as profiles have to remain consistent with the semantic defined by the UML meta-model (Selic, 2007). As such a profile cannot be used to define a new meta-model. Rather, the objective of profiles is to offer a straightforward mechanism to adapt the UML meta-model with constructs of a particular domain. Figure 1.4 displays the core concepts of UML profiles as defined within the UML meta-model.

An UML Profile is a specialization of the UML Package. A profile defines a number of stereotypes which add non-standard semantics to the model elements on which they are applied. Stereotypes are classes that extend base meta-classes. They may include properties and may be accompanied by constraints enforcing rules that are applicable to the stereotypes. To define such constraints, the OMG provides the Object Constraint Language (OCL) (OMG, 2014b). Figure 1.5 provides a simplified example of an UML profile for Enterprise JavaBeans (EJB). The profile defines the abstract stereotype Bean that is required to be applied to the Component metaclass. In other words, it means that an instance of either the Entity or Session stereotype must be applied to each instance of Component. Furthermore this profile defines constraints to verify that models are well formed. Example of such constraint include that a component should not be generalized or specialized.

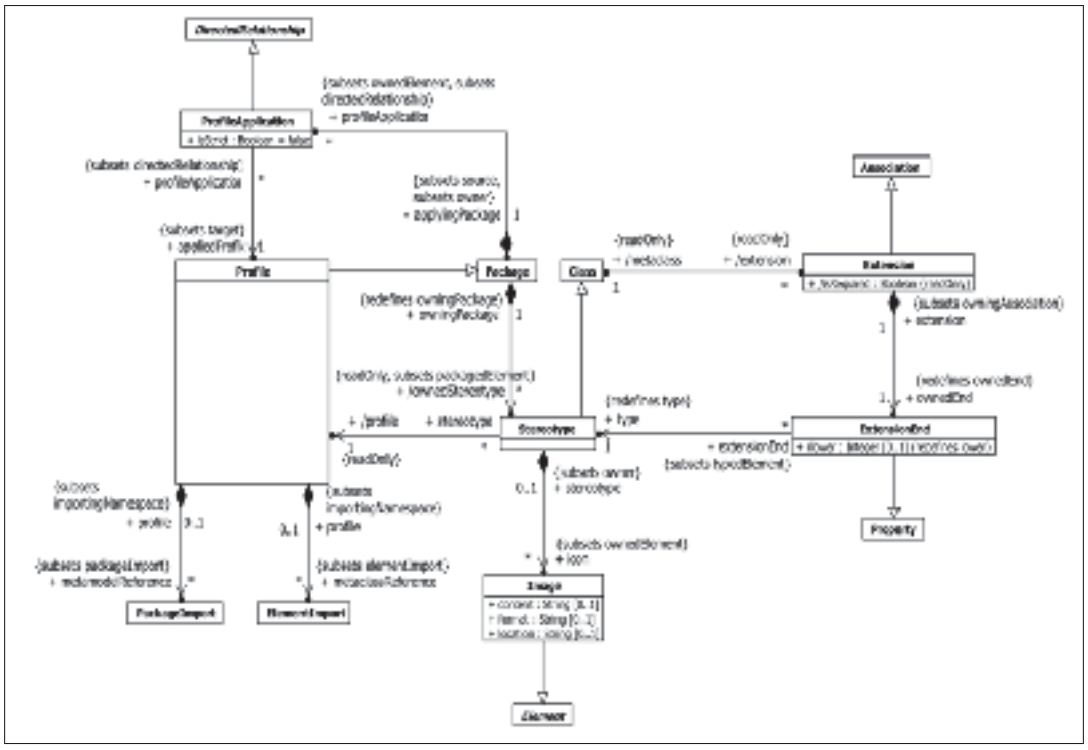


Figure 1.4 UML meta-model: Profile mechanism definition (OMG, 2015).

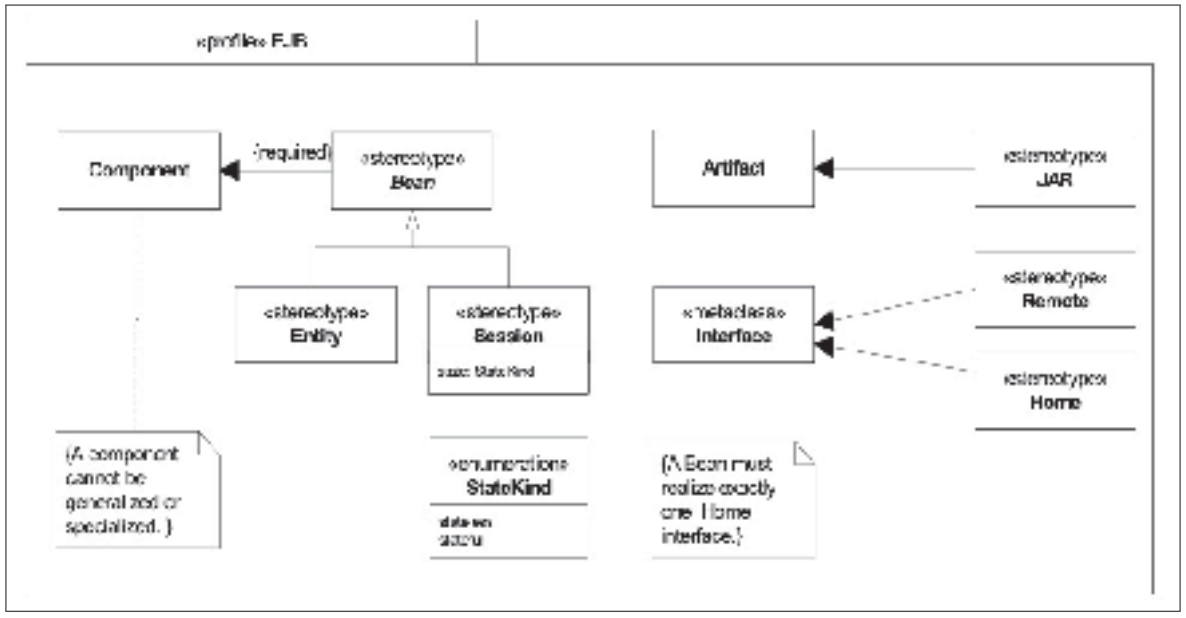
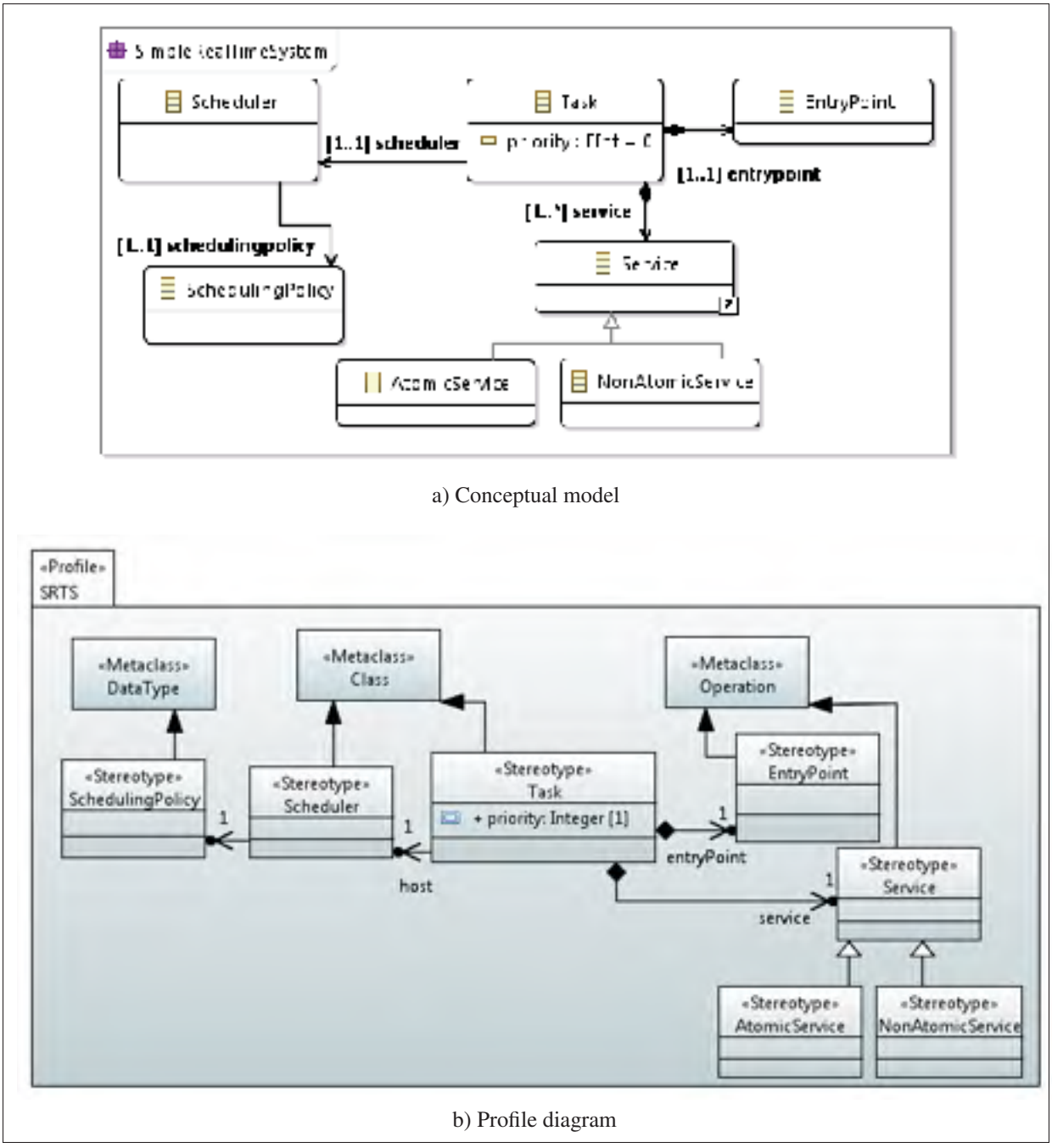


Figure 1.5 Example of UML profile for EJB. (OMG, 2015).

Despite being a rather simple mechanism to create domain specific modeling language, there exist no standardized methodology to guide in the design of UML profiles. However study of various profiles revealed an approach that is common to build an UML profile. Selic (2007), Lagarde *et al.* (2008) and Fuentes-Fernández & Vallecillo-Moreno (2004) describe this process to design UML profiles in a similar manner. The general approach to define UML profiles shall be composed of the following steps: 1) the profile designer with help from domain specialist defines the conceptual domain model, 2) the profile designer realizes a transformation of the domain's concepts into stereotypes by mapping the domain concepts to the appropriate UML meta-classes and 3) the profile is reviewed to verify its consistency against the UML meta-model.

Figure 1.6 provides a small example of the mapping of a conceptual model into an UML profile. The conceptual model introduces concepts for a Simple Real Time System (SRTS). A Task represents any resource that can be scheduled, it contains a reference to one Scheduler that shall be defined by its SchedulingPolicy. Furthermore a Task has an EntryPoint and has a set of services (atomic and non atomic). The resulting profile is defined by creating a stereotype for each of the defined concepts. Concepts Scheduler and Task are extending the Class metaclass. SchedulingPolicy extends the DataType metaclass. Finally, Service and EntryPoint extend the Operation metaclass.



a) Conceptual model

b) Profile diagram

Figure 1.6 Example of the mapping of a conceptual model into a UML profile. Adapted from Lagarde *et al.* (2008)

1.3 Approaches for modeling safety critical systems

Many model-based approaches have been proposed recently to support the development of software in the context of safety-critical systems. Although our research problem is particularly aimed at airborne software and their development according to DO-178C, we have explored a broader range of application domains for model-based approaches, extending the scope of our literature review to approaches pertaining to the development of safety-critical software in general. The reason behind this wide scope, is due to the fact that safety-critical software share similar properties and challenges independently from their domain of application such as railway, aerospace, energy and medical devices.

Thus, we first present the approaches that are domain-independent in Section 1.3.1. Then, we introduce the approaches that are domain-dependent in Section 1.3.2. Specifically we introduce UML profiles that target various specific safety-critical systems and those that specifically target avionic systems.

1.3.1 Domain-independent approaches

Generally, safety compliance is not based on just one standard but a corpus of regulatory standards. In this context, De la Vara *et al.* (2016) introduce the Reference Assurance Framework (RAF) metamodel. Its purpose is to express key concepts and relations used for demonstrating safety compliance that are extracted from multiple sources (i.e. safety standards, specific domain recommended practices, and company specific practices). The RAF meta-model provides an unified mean to create models used for safety assurance and certification. An excerpt of a RAF model for IEC 61508 is provided in Figure 1.7 and depicts the use of some of the main concepts of the RAF meta-model. Among these concepts, `ReferenceRequirement` captures conditions that might have to be fulfilled, a `ReferenceActivity` defines activities that must be executed. These activities produce `ReferenceArtifact` that are the data that must be managed and provided for certification. A `ReferenceTechnique` specifies the way an activity is performed or artifacts are created.

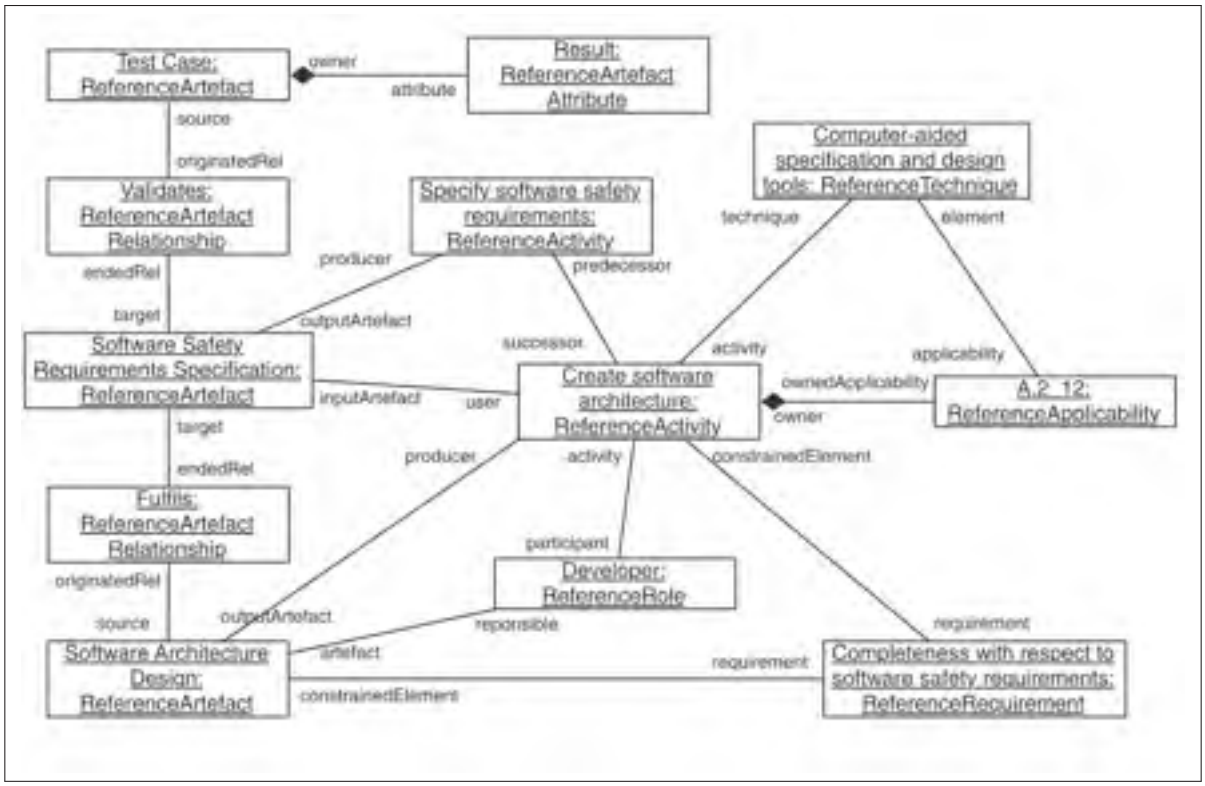


Figure 1.7 Excerpt of a RAF model for IEC 61508. Adapted from De la Vara *et al.* (2016).

Safety standards rely on the traceability of safety evidence throughout the complete software life cycle to both demonstrate compliance with standard and support claims about the safety of the software product. Commonly required safety evidence includes: test cases, test results, and system specifications (requirements). Because suppliers must collect and maintain these evidence, the explicit specification of the traces between these artifacts is an important aspect to support the certification process of safety critical systems.

Work from Nair *et al.* (2014) introduces a Safety Evidence Traceability Information Model (SafeTIM). Its objective is to provide a broad overview of safety evidence traceability in the context of safety critical systems. The proposed model captures the traces and evidence information that must be created and maintained to show compliance with safety standards. SafeTIM was developed based on an extracted set of traces that are necessary for safety evidence. As observed on Figure 1.8, the principal concept that traces to all of the concepts of the model is

the Artefact concept, which represent an individual and identifiable unit of data that is managed throughout the software life cycle. These artifacts are used as piece of evidence for claims, which are propositions that are being asserted in relation to system safety. Those piece of evidence are accompanied by arguments, which are body of information that are provided in order to establish a claim about the system safety. Artefacts are the output and are also required as input data for various activities of the software life cycle.

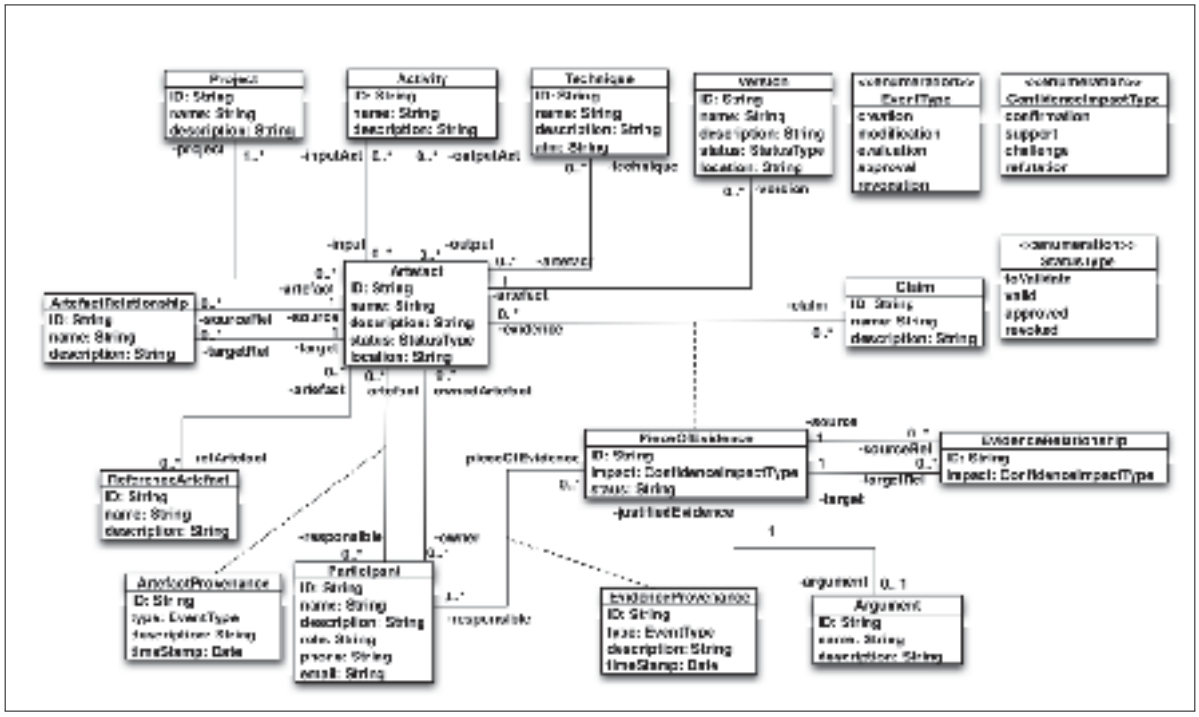


Figure 1.8 The Safety Evidence Traceability Information Model (SafeTim). Adapted from Nair *et al.* (2014).

Nejati *et al.* (2012) introduce a SysML based approach to address traceability between safety requirements and their design implementation. An algorithm that analyses the association between a requirement and its implementation is provided to extract design slices. Design slices provide a detailed view of the system from the perspective of a specific safety requirement. These are extracted from the overall design and capture the design aspects related to a target requirement. Such slice enables the analysis of the implementation of a safety requirement by removing the design elements that are irrelevant for the requirement under analysis. Figure 1.9

provides an example of the resulting design slice that is extracted from the design implementation of a provided requirement.

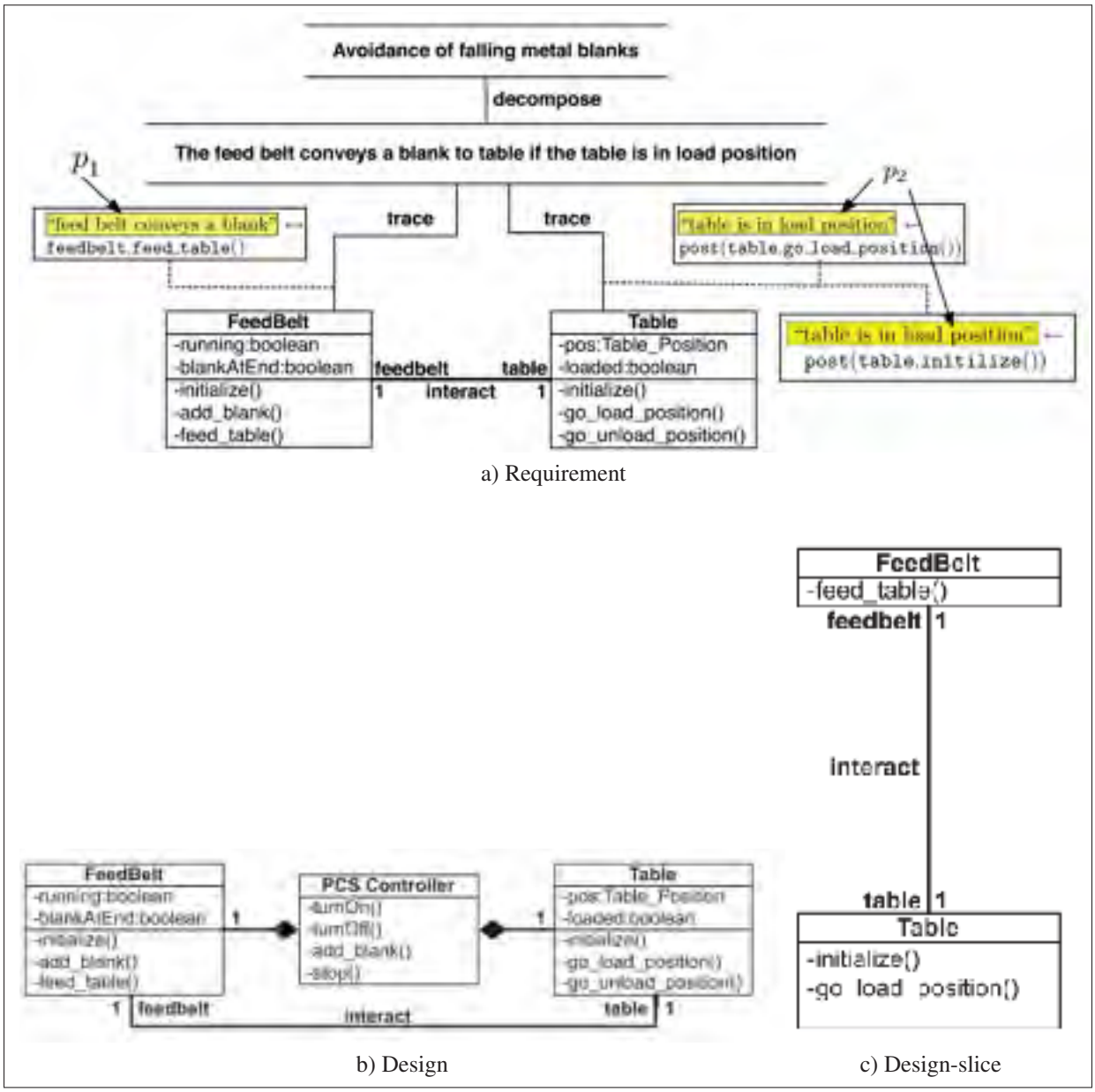


Figure 1.9 Example of a requirement (a) and its related design slice (c) that is extracted from a design model (b). Extracted from Nejati *et al.* (2012).

1.3.2 Domain-specific approaches

1.3.2.1 UML profiles targeting various specific safety critical systems

Berkenkötter & Hannemann (2006) propose a domain specific language in the form of a UML profile for the railway control systems domain (RCSD). The RCSD profile enables the precise modeling of the static description of railway networks and their associated dynamic aspects. Networks are comprised of elements such as track segments, points, signals, and sensors. These elements are the physical entities that constitute a network of tracks on which trains are moving through pre-defined routes.

The profile models the domain using a combination of class diagrams and object diagrams. Class diagrams are used to represent problems of the railway domain (i.e. tramway and railroad models) whereas object diagrams capture instances of these problems (i.e. the explicit track layout). The object diagram uses either the UML notation or a notation introduced by the authors based on the symbology of the railway domain. Figure 1.10 shows an overview of an object diagram using the specific notation for the railway domain introduced by the RCSD profile (left side of the figure) and the UML notation (right side of the figure).

Figure 1.10 Object diagrams representing two track segments along their sensors and signals using both UML and RCSD notation. Extracted from Berkenkötter & Hannemann (2006).

The dynamic aspect of the track network is defined as a timed state transition system (TSTS). The timed transitions are embedded locally in the profile's elements. To ensure safety through the network, a controller is defined and added to the network model and remain independent from he physical elements captured in the model. The controller includes the safety conditions for running the systems. The controller model is defined using a strict mathematical model. This mathematical definition enables to prove the violation of the safety conditions for the running system by using bounded model checking techniques.

Panesar-Walawege *et al.* (2013) and Kuschnerus *et al.* (2012), both defined UML profiles aimed at the expression of certification-related information for IEC 61508 standard. In particular, Panesar-Walawege *et al.* (2013) proposed an approach to support safety-critical suppliers in creating safety evidence needed to show compliance with a specific safety standard. The approach is based on a process that assists preparing of the certification evidence. This process is comprised of 4 phases as shown in Figure 1.11. The first two phases, occurring only once per targeted standard, are similar to the methodology described in the work of Lagarde *et al.* (2008) and Selic (2007) for defining UML profiles. The first phase consists in the definition of the conceptual model that captures the concepts of the standard under scrutiny related to certification evidences. The second phase, consists in the mapping of these concepts to the UML meta-model to obtain an UML profile. The third phase of this approach is the application of the profile to the domain model of the system undergoing certification. The resulting model allows the capture of precise links between the system's concepts and standard's concepts. In the last phase, the resulting model is instanciated to create evidence submitted for certification.

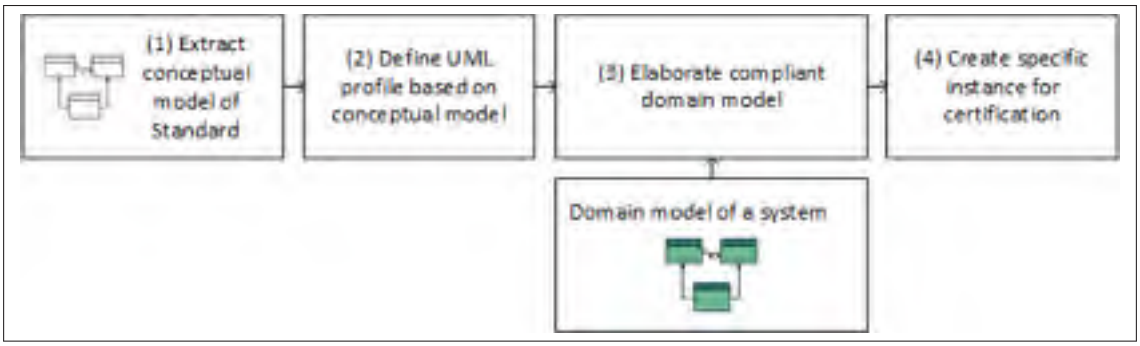


Figure 1.11 The process for the creating evidence of a safety standard. Extracted from Panesar-Walawege *et al.* (2013).

Kuschnerus *et al.* (2012) introduced a UML profile based on the concepts extracted from IEC 61508. The profile uses models as baseline artifacts for certification documentation. The process to extract the domain model and its mapping to the UML meta-model that defines the profile is similar to the methodology described by Lagarde *et al.* (2008). The domain model extracted from the standard is divided into two categories. The first category of concepts is related to the definition of the safety process defined by IEC 61508 and captures the activities and recommended techniques as defined by the standard. This category includes the definition of monitoring concepts for the process of designing the architecture. The second category of the domain model defines concepts that are specific to the standard such as safety terms and their relations. This include safety functions and the certification status of software modules. An excerpt of the domain model is provided in Figure 1.12.a. It contains concepts related to the first part of the domain. It defines the relations between an Electrical/Electronic/programmable Electronic(E/E/PE) safety-related system, the safety integrity the system needs to conform with and the techniques that are performed in order to realize the system. SIL Activity represents activities defined by the standard and each of these should use one or more techniques depending on the targeted integrity level. Figure 1.12.b provides communication concepts of the second part of the domain. In a safety-critical system, a communication shall either be safe (i.e. transmission of data is verified by a checksum) or unsafe. Communication is established between multi-layered communication stack using a channel as a medium. Each channel defines the protocol it uses in order to transmit data. This profile focused mainly on safety requirements.

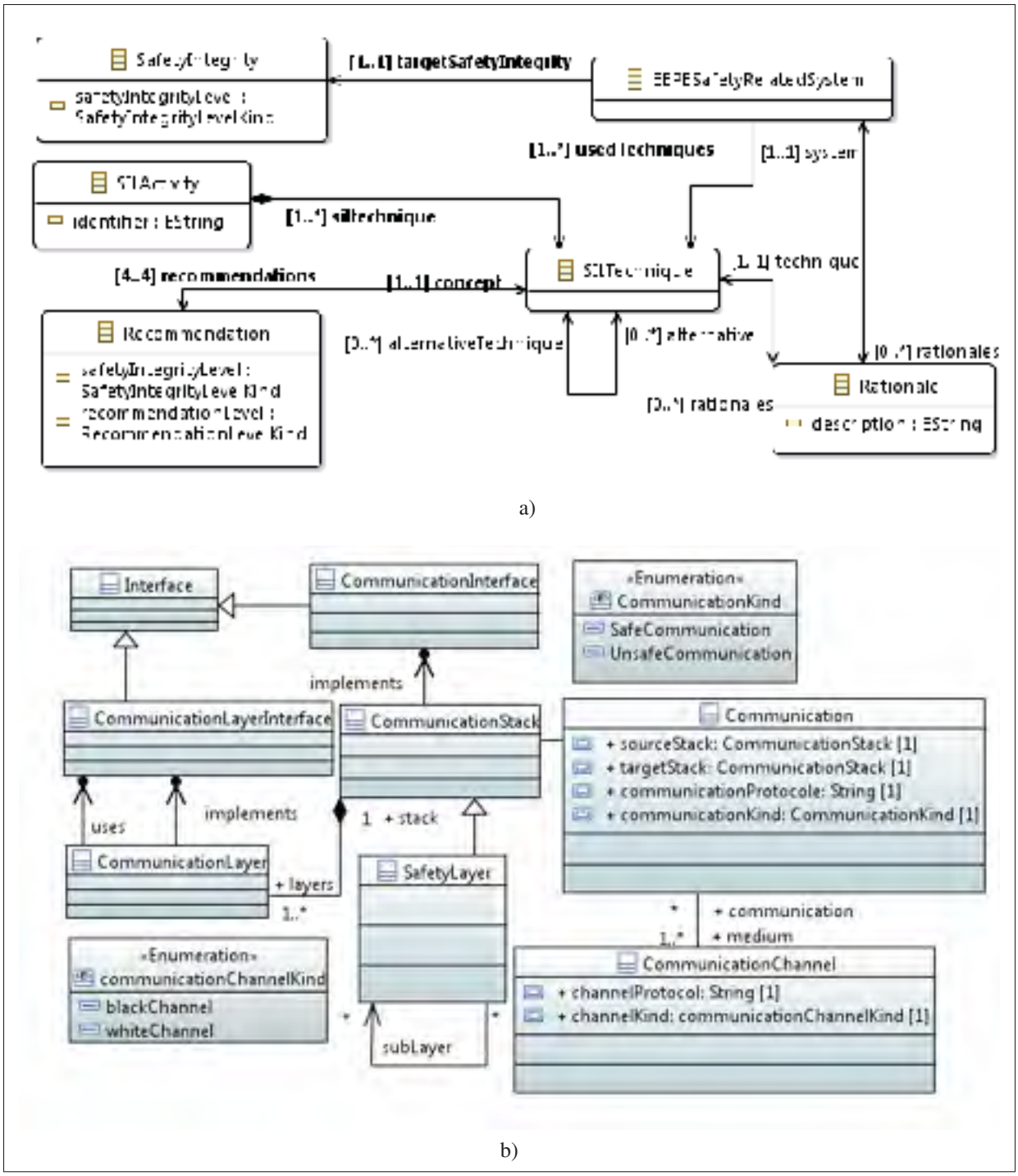


Figure 1.12 Excerpt of Kuschnerus et al.'s domain model. Adapted from Kuschnerus *et al.* (2012).

Safety-critical systems behavior are often dependent on various timing properties, thus the correct timing of such systems is part of their functional correctness. In this context, the UML profile for Modeling and Analysis of Real-Time Embedded System (MARTE) (OMG, 2011) primary concern is to capture the aspects related to real-time in embedded systems. MARTE is structured as a hierarchy of sub-profiles, as provided in Figure 1.13.

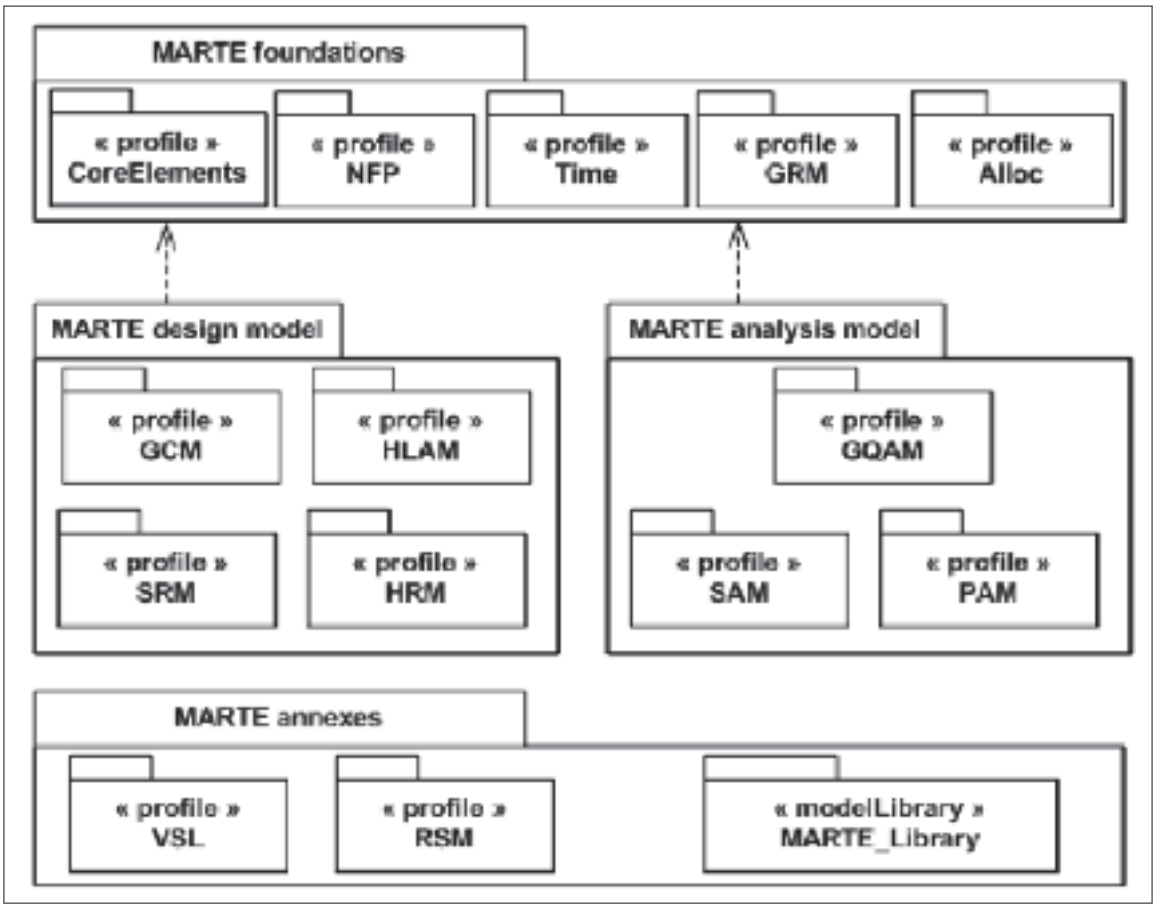


Figure 1.13 Architecture of the MARTE profile (OMG, 2011).

The "Marte foundations" package defines the foundation on which the rest of MARTE is built. It defines four basics sets of extensions to UML, these sub-profiles are the following:

- **Non-functional properties (NFP)**: provides modeling constructs for declaring, qualifying, and applying semantically well-formed non-functional aspects of UML models. It is

completed by the "Marte annexes" sub-profile Value Specification Language (VSL) which is a textual language used for declaring algebraic expressions.

- **Time**: provides the concepts for defining time in applications and for manipulating its underlying representation.
- **Generic resource modeling (GRM)**: provides an ontology of resources enabling the modeling of common computing platforms (i.e. resources on which an application is allocated for computation) along with the concepts needed for specifying resources usage.
- **Allocation modeling (Alloc)**: provides the concepts pertaining to the allocation of functionalities to the entities responsible for their realization. These concepts may be either time allocation (i.e. scheduling) or spatial allocation (i.e. hardware allocation).

The remaining parts of MARTE are separated into two categories of extensions: "MARTE design model" and "MARTE analysis model". Design models are created using annotations containing concerns from real-time or embedded systems that are provided by the High-level Application modeling (HLAM) sub-profile. Also MARTE allows the modeling of component based systems through its Generic Component Model (GCM) sub-profile. Analysis models are created using the Generic Quantitative Analysis Modeling (QGAM) and its two refinements dedicated to both schedulability (SAM) and performance (PAM) analysis.

1.3.2.2 UML profiles for avionics software

Wu *et al.* (2015) have developed a methodology called Safety Oriented Architecture Modeling (SOAM). The method focuses on the design of a component centric architecture for avionic software in the context of DO-178C. More precisely this approach emphasizes on the notions related to the safety of software components. The method introduces an UML profile named *SafetyProfile*. Authors claim that the profile captures safety properties in accordance with DO-178C guidelines that apply to software components and their related interfaces. Figure 1.14 presents an excerpt from the conceptual model from which the profile was derived.

The profile focuses on components-based architecture design. In fact the conceptual model focuses on the communications between components, the definition of the component's interfaces and their monitoring. The `SafetyComponent` is the main concept of this model. A `SafetyComponent` communicate with another through a `SafetyChannel`. A component may detect a `Fault` and needs to handle it through various `MitigateAction`. A component defines `SafetyInterface` that are accessed throught its defined `SafetyPort`.

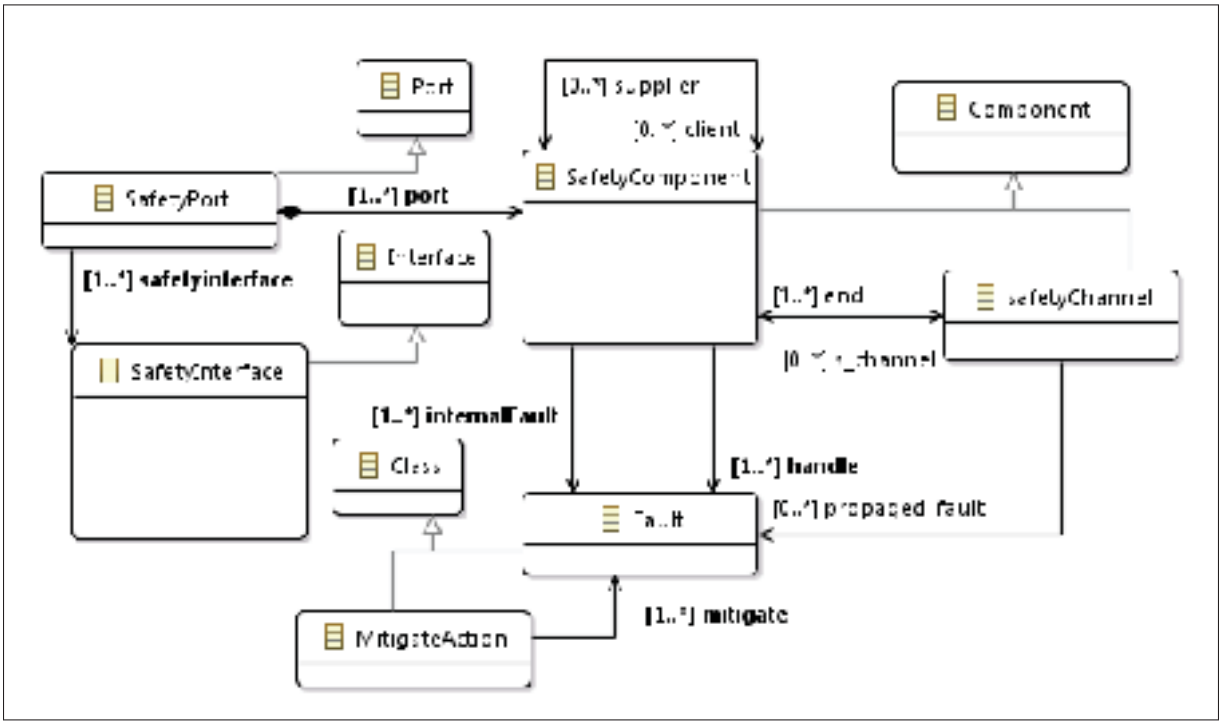


Figure 1.14 Excerpt of the conceptual model used for building SafetyProfile. Adapted from Wu *et al.* (2015).

Zoughbi *et al.* (2010) introduced SafeUML, a UML profile based on DO-178B. Its purpose is to capture the safety related requirements that are allocated to software and to monitor their implementation through the software design. Furthermore the profile intends to improve communication and collaboration between safety engineers, software engineers and the certification authorities. The profile is organized into packages, each package includes a set of related concepts. The concepts from which the profile is developed are grouped into five packages: 1) the *Requirements* package contains the concepts that are needed to express software require-

ments, their refinement as well as the traceability of the requirements to design artifacts, 2) the *Characteristics* package contains concepts to identify design elements having a direct impact on safety by specifying the software level that is attached to these elements along with the failure conditions that are associated to these elements, 3) the *Event Management* package that defines the concepts of events and the actions related to their capture, 4) the *Configuration* package that defines the concepts to capture elements related to software configuration, user modifiable software and change control, and 5) the *Replication* package containing concepts to address software redundancy. Figure 1.15 provides an overview of three of the packages that constitute this conceptual model. A Partition is created to fulfill one or more Requirements, and is partitioned from one or more SafetyCritical entities. A SafetyCritical entity may trigger Events that must be monitored in order to be detected in the system. The Monitor is in charge of notifying various Handlers that perform Reactions associated to the captured event.

Although DO-178B does not provide guidelines for the use of model based software development and verification, Stallbaum & Rzepka (2010) introduced a UML profile to enable the specification of DO-178B compliant test models. The purpose of these test models is twofold. The first purpose is to enable testing activities of the software as per DO-178B guidelines. The second is to enable the use of the models as artifacts supporting evidence for the certification process by capturing the required relevant testing information. From the analysis of the standard, they identified the information that test models must capture and designed an UML profile, which is depicted in Figure 1.16. The profile is used by applying the TME (TestModelElement) stereotype to each model element to define test and certification relevant information. A TME is the entity that represents a test model element. It comprises the system behavior and certification related information. Examples of TME include activity, interaction or state. The Requirement stereotype is used to specify software requirements and the association between requirements and TMEs supports requirement-based test coverage analysis. The SafetyRationale stereotype specifies whether or not an element of the model is safety-critical and includes the rational for the element's criticality. The Interface stereotype is used to specify hardware/software interfaces. The SoftwareComponent stereotype defines self-contained unit of the software that implement

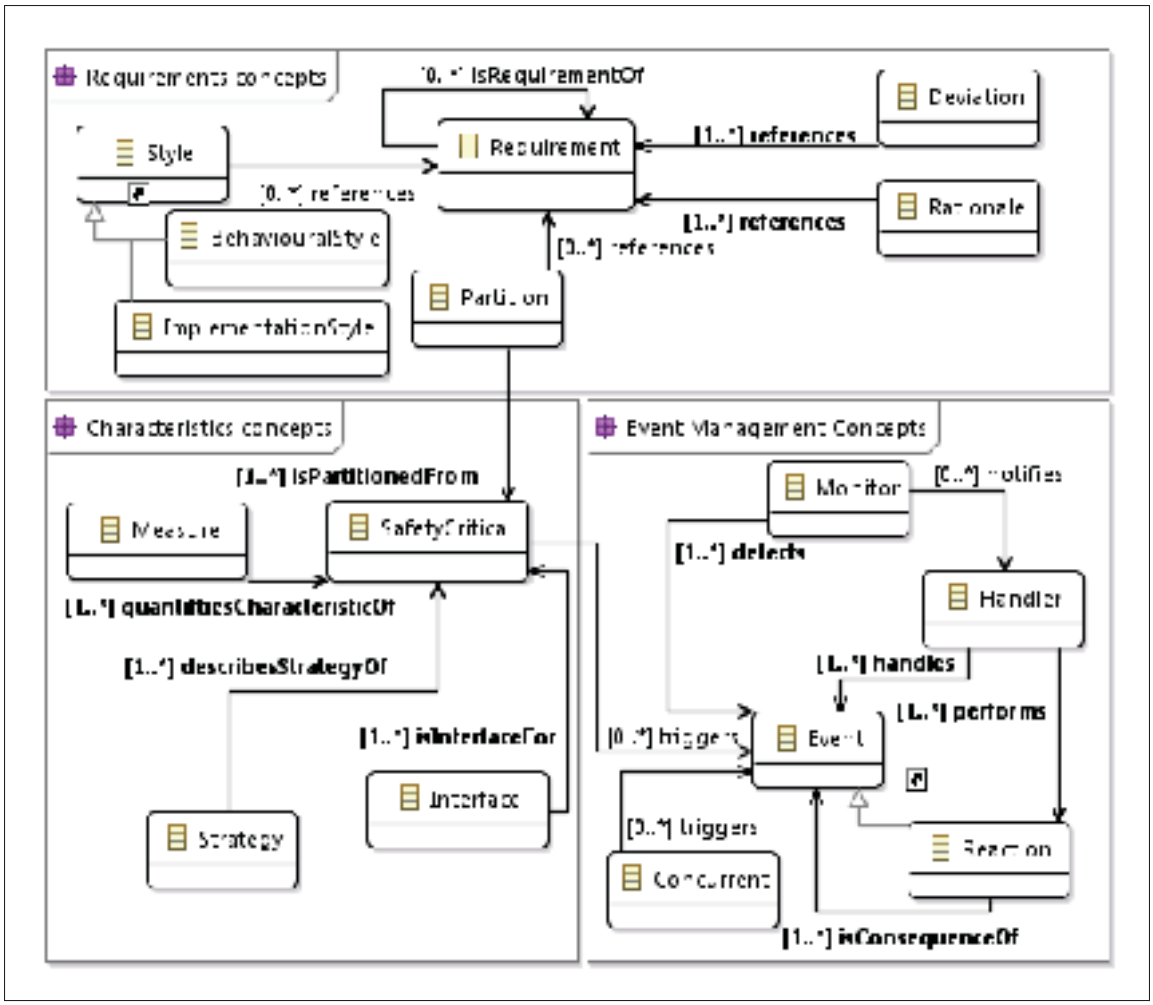


Figure 1.15 Excerpt of Zoughby *et al.*'s safety-related conceptual model. Adapted from Zoughbi *et al.* (2010).

distinct functionality of the system. The profile enables the capture of the following testing needs requested by DO-178B: (1) traces between model elements and their related requirement(s), (2) identification of the software level(s) along the corresponding rationale, (3) identification of the conditions related to normal range and robustness test cases, (4) identification of the testing method (i.e. hardware/software integration, software integration, and low-level tests), (5) identification of hardware/software interfaces and their parameters, (6) traces between model elements and software components, (7) traces between model elements and source code, and (8) the type of the traces.

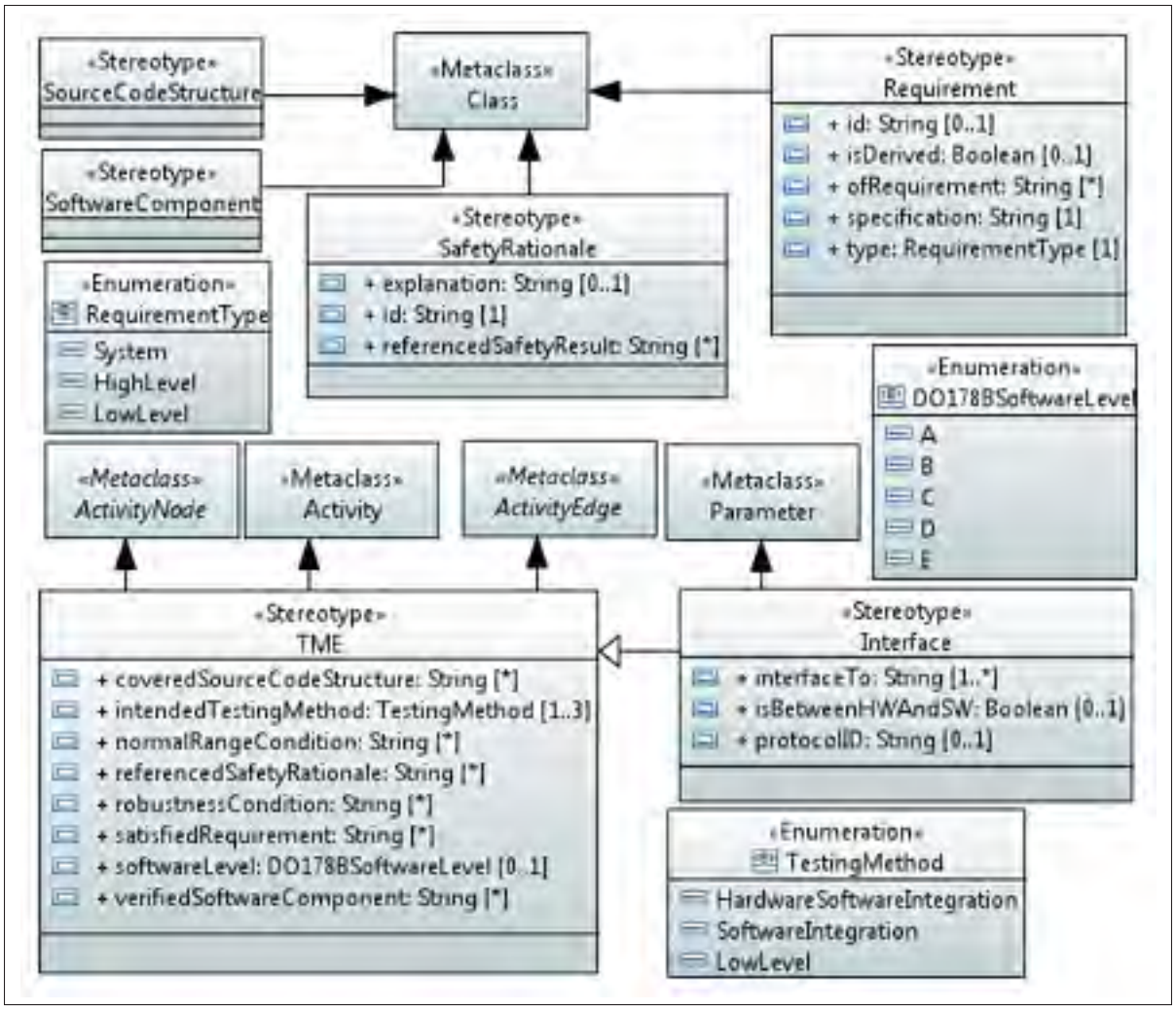


Figure 1.16 UML profile for DO-178B compliant test models. Adapted from Stallbaum & Rzepka (2010).

1.4 Discussion

The studied approaches contribute in different ways to support the development and certification of safety-critical software. Table 1.2 provides an overview of the domain of application, the objectives, the targeted standard and the software life cycle/ concerns of the approaches studied in Section 1.3.

Table 1.2 Summary of the studied approaches to model various safety critical systems.

Approach	**Domain**	**Objectives**	**Targeted standard**	**Software life cycle processes/ Concerns**
De la Vara *et al.* (2016)	Multiple domains	Express key concepts and relations used to demonstrate safety compliance extracted from multiple standards	Multiple standards (including DO-178C)	Safety assurance and certification
Nair *et al.* (2014)		Provide a model of safety evidence traceability		Traceability
Nejati *et al.* (2012)		Extract design slices related to a requirement		Traceability of requirements to design
Berkenkötter & Hannemann (2006)	Transportation: Railway, Subway	Model the static and dynamic aspects of the domain		Software Design
Panesar-Walawege *et al.* (2013)	Electrical Electronic Programmable Devices	Support the process of collecting evidence for the support of certification against a standard	IEC 61508	Safety Requirements
Kuschnerus *et al.* (2012)	Electrical Electronic Programmable Devices	Model the standard	IEC 61508	Planning
Object Management Group (OMG) (2011) Profil: MARTE	Reel time embedded software	Capture and analyze real time properties of a software		Specification of requirements and design

Approach	**Domain**	**Objectives**	**Targeted standard**	**Software life cycle processes/ Concerns**
Wu *et al.* (2015) Profil: SafetyProfile	Avionic	Capture safety related aspects assigned to a software component	DO-178C	Software Design Process
Zoughbi *et al.* (2010) Profil: SafeUML	Avionic	Capture software requirements and monitor their implementation	DO-178B	Software Requirements Process, Software Design Process
Stallbaum & Rzepka (2010)	Avionic	Capture test data as UML models	DO-178B	Software Verification Process

Several of the explored approaches emphasize the possible ambiguous interpretation of textually defined safety standards. Because of the need to provide evidence that a standard was adequately applied for the development of certifiable software, an ambiguous interpretation of the standards results in major risks for certification. Misinterpretation of a standard may most possibly result in the creation of inadequate evidence hindering an already difficult certification process. This issue calls for a unique interpretation of the standards. As a result, the introduction of model-based development may help in reducing or suppressing the possible misinterpretations by offering the capability to model the standards. Thus the studied approaches provide an explicit interpretation of the standard through the models they propose. Furthermore modeling of safety standards may lead to an automation of certain aspects of the production of artifacts supporting certification.

Many approaches specifically tackle avionics software (e.g. Wu *et al.* (2015); Zoughbi *et al.* (2010); Stallbaum & Rzepka (2010)), however they propose meta-models and profiles that target specific concerns in the software development. Stallbaum & Rzepka (2010) target the software

verification process, while Wu *et al.* (2015) target the software design process by capturing safety requirements that apply to component-based software architecture. Finally, Zoughbi *et al.* (2010) target the software requirement and design process by capturing various safety-related information within the models. These approaches do not model the DO-178 standard. The RAF meta-model (De la Vara *et al.*, 2016) is an exception as it is built from a number of standards, including DO-178C. However it uses a unified vocabulary that is not specific to DO-178C.

Moreover, the existing approaches do not tackle the assurance level modularity introduced by DO-178. As a result these approaches do not consider the variations in the compliance needs to be provided in order to achieve certification. Indeed, the standard offers its guidelines in an adaptive manner against the targeted design assurance level.

DO-178 also describes the specific objectives and activities to be performed along with the resulting evidence to be produced during the software life cycle. None of the explored approaches captures the information related to the software planning process as required by DO-178C. The plan for software aspects of certification (PSAC) captures information about the methodologies and techniques that are used to develop the software product. These information are important as they enable the certification authorities to state whether or not the applied methodologies and techniques are considered to be sufficient to provide confidence in the produced safety evidence.

D0-178C introduces a number of traceability requirements and needs. This introduces a number of traceability concepts that need to be modeled. These concepts are partially covered by existing approaches as they respectively target different aspects of the software life cycle. The approach in Nair *et al.* (2014) proposes a more complete taxonomy of traceability but the proposed taxonomy is generic and does not specifically target DO-178.

As some of these approaches were released before the current version of DO-178 they simply could not address the guidelines introduced in the supplements to DO-178C. For instance, among these new guidelines, in the context of model based development, DO-331 enforces the separation of specification models and design models. However, some of the approaches capture the requirements directly into the design models.

These limitations led us to define our own domain-specific language, based on the use of UML profile, to support DO-178C and its supplements.

CHAPTER 2

PROPOSED APPROACH AND METHODOLOGY

In this chapter, we first introduce our research objectives (Section 2.1). In Section 2.2 we provide an overview of the proposed approach in order to achieve theses objectives. Finally in Section 2.3, we describe the methodology that guided our research.

2.1 Research objectives

The general goal of our work is to support the process of collecting the information that is used for airworthiness certification. In particular, our goal is to benefit from the advances made in model-driven technologies to specify the evidence used to achieve software certification in the context of airborne systems that fall under the regulatory scope of DO-178C. In this context, our specific objectives are:

- Study and analyze DO-178C to identify the information required as evidence for certification.
- Propose a model-driven approach that supports the specification and management of the certification evidence taking into consideration the assurance level of the software under development.
- Implement and assess the proposed approach.

In this research, we focused on specific parts of the software life cycle, namely the software planning process, the software requirements process, the software design process and the software verfication process. Moreover we limited the scope of our analysis to DO-178C guidelines and two of its supplements: DO-331 ("Model-Based Development and Verification Supplement to DO-178C and DO-278A") and DO-332 ("Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A"). This is due to the fact that the industrial partners were using model-driven and object-oriented technologies.

2.2 Proposal: An assurance level sensitive UML profile to capture DO-178C relevant certification information

To achieve our research objectives, we propose to build a domain-specific modeling language that captures information required as evidence for DO-178C certification. The proposed language should enable the specification of the software life cycle data in terms of DO-178C objectives and activities for each software process. The language should also support the specification of DO-178C data related to software requirements, design and verification. Moreover, the language must provide means to capture traceability between requirements, design and verification data as required by DO-178C.

An important aspect of the proposed language is to ease the collection of certification evidence according to the software level. In fact, as discussed in the previous chapter (Section 1.1), DO-178C defines five software levels (also named design assurance levels) that are mapped to different failure conditions ranging from "catastrophic" (software level A) to "no safety effect" (software level E). Each software level requires different DO-178C activities to be performed and DO-178C objectives to be satisfied. Thus the proposed language is an assurance-level sensitive modeling language. In other words, our domain specific modeling language introduces a number of constraints which ensure that the required information for certification is specified according to the software level. Thus these constraints are expressed in terms of DO-178C concepts captured by the language (e.g. activities, traceability data, verification data) and in terms of the software level. Figure 2.1 illustrates the concept of an assurance-level sensitive modeling language.

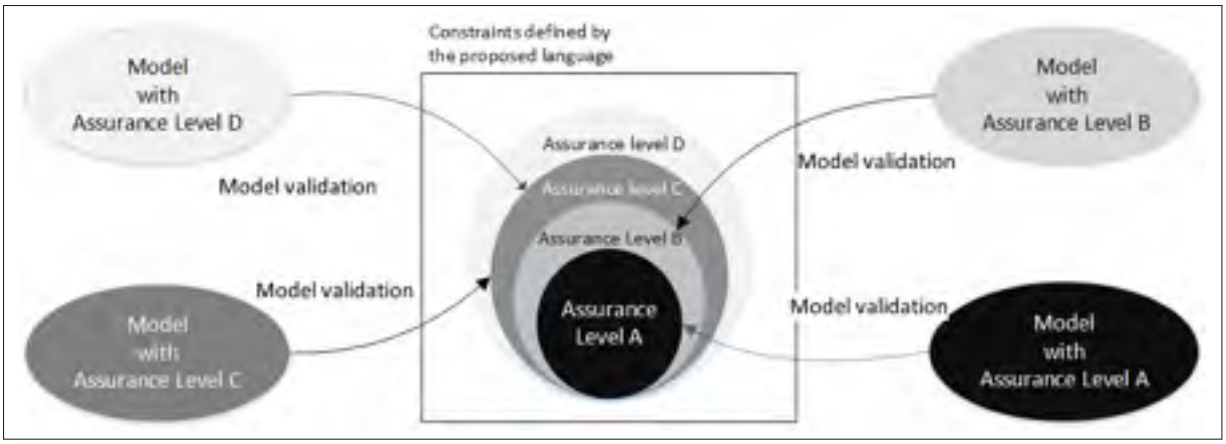


Figure 2.1 Constraints used for model validation against a designated assurance level.

We implemented our domain specific modeling language using the UML profile technology. The choice for defining this modeling language as an extension to UML through its profile mechanism has been motivated by the following reasons:

- UML is one of the languages that is actually used by our industrial partners for their model-based development activities.
- UML has been defined so that it could be extended with concepts from specific domains of application through its profile mechanism. We provided a description of the UML extension mechanisms in Section 1.2.2.
- The number of available modeling tools that support UML and its profile mechanism.

2.3 Research methodology

In order to achieve our research objectives, we adopted a four-phase research methodology as described in Figure 2.2.

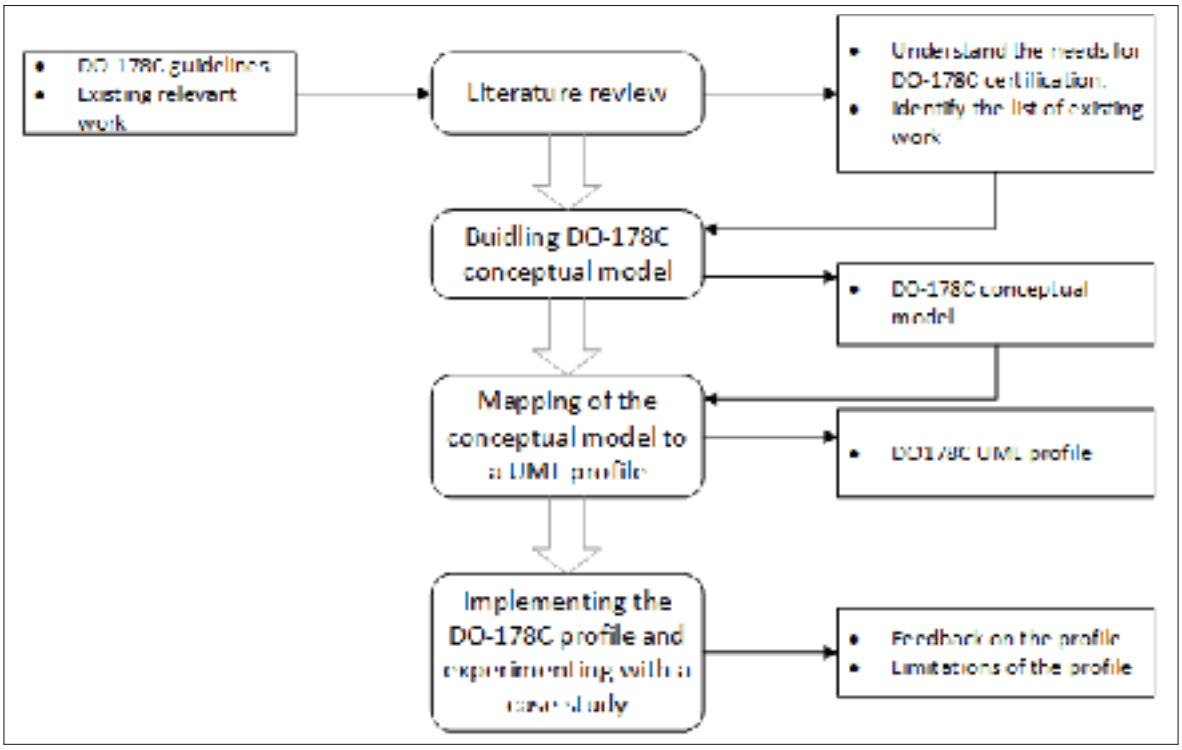


Figure 2.2 Phases of the research methodology

Phase 1: Literature review

The first phase of our research methodology consists in studying the literature related to the development of safety-critical software based on model-based approaches. Our literature review has been performed with the purpose of:

- ***Obtaining valuable knowledge of DO-178C***: We studied DO-178C to obtain extensive knowledge of the standard's content and the constraints it imposes on the process of developing airborne software. This comprehensive review of DO-178C enabled us to go forward in the process of modeling the concepts defined by DO-178C.
- ***Understanding the needs of safety critical systems***: To understand the issues related to safety-critical software development, we studied a number of existing software development approaches, especially model-based approaches targeting safety-critical software. The study of these approaches led us to identify their limitations in supporting DO-178C certification.

Phase 2: Building DO-178C conceptual model

During this phase, we further analyzed DO-178C in order to build a conceptual model that captures the concepts embedded in the standard.

Phase 3: Mapping of the conceptual model to a UML profile

The third phase of our methodology takes as input the conceptual model resulting from the second phase of our methodology. During this phase, we perform the mapping of the concepts we were able to extract from the standard to the UML meta-model. This activity results in the creation of an UML profile.

Phase 4: Implementing the DO-178C profile and experimenting with a case study

During this phase, we first implement our profile within an open-source UML modeling tool. We then assess our profile by using it to specify a realistic case study; i.e. the landing gear control software (LGCS) (Paz & El Boussaidi, 2017). In particular, we used the profile to carry out four use-cases corresponding to the objectives targeted by our domain specific modeling language as discussed in Section 2.1. These use cases are the following:

- Specify the software life cycle data according to a design assurance level.
- Specify the software requirements: The profile should provide modeling constructs to specify system-requirements allocated to software and their subsequent refinement into high-level and low-level requirements as required by DO-178C.
- Specify traceability between requirements and design: Using the profile, we should be able to trace high-level requirements to software design. In the context of DO-178C, software design includes software architecture and low-level requirements.
- Specify verification data: The profile should provide constructs to specify verification data including reviews, analyses, test cases and procedures. Moreover, the profile should provide

means to ensure the traceability between the verification data and software requirements and design data.

CHAPTER 3

A DO-178C CONCEPTUAL MODEL

From the performed analysis of DO-178C and its supplements, we have extracted a conceptual model from which the proposed profile has been developed. As discussed in Section 2, we limited the scope of our analysis of DO-178C to a subset of processes. These processes include the software planning process, the software requirements process, the software design process and the software verification process. The concepts in the resulting conceptual model, are divided into groups corresponding to the analyzed DO-178C processes. Thus this chapter is organized as follows. Section 3.1 introduces the template used to describe each concept of the conceptual model. In Section 3.2 we describe the concepts related to the software planning process. In section 3.3, we introduce the concepts related to the software requirements process. Finally in Section 3.4, the concepts related to the software verification process are defined.

3.1 The template for describing the conceptual model

To describe the concepts that are part of the proposed conceptual model in a consistent and uniform way, we used a template. The template contains the following sections:

Definition

This section provides the definition of a concept. In particular, it describes the concept and its purpose.

Generalizations

Provides the list of concepts that are specialized by the the specified concept. This section is provided using a table as follows:

Parent concept
Parent concept

Attributes

This section provides the list of attributes for a concept. Attributes are used to capture relevant properties and characteristics related to a concept. A name and a description are provided for each attribute. The list of attributes is provided using a table as follows:

Name	Description
targetedSoftwareLevel	Indicate the project's targeted software levels. Because a project might be composed of more than one component, each of them might be assigned a different software level.

Relationships

This section provides the list of the relationships that a concept has with other concepts. For each relationship, a name and a description are provided. The list of relationships of a concept is provided using a table as follows:

Name	Description
derivedBy	Identifies the requirements, 0 or more, that derive the specified requirement.

Constraints

This section provides the list of constraints that applies to the specified concept. The constraints are defined by software levels.

The software level for a constraint represents the minimum software level for which the constraint shall be verified. In other words, if a constraint is defined for software level D, then the constraint also applies to software levels A, B and C. Moreover, each DO-178C supplement may introduce additional constraints.

We do not enforce the use of a specific language to specify the constraints due to possible implementation limitation of tools.

Constraints are provided using a table as follows:

Constraint description	**Software level**	**Introduced by supplement**
A requirement *id* must be unique	Software level D	None

3.2 Software Planning Process

During the software planning process, an applicant defines the activities that have to be carried out within the software life cycle processes. The software planning process also defines the software life cycle in terms of the sequencing of the processes, the transition criteria between them and the feedback mechanisms (RTCA, 2011a). In particular, the plan for software aspects of certification (PSAC) produced during the planning process, is submitted to the certification authority to asses whether or not the application of a defined software life cycle may result in the production of evidences that are deemed adequate to demonstrate the safe properties of a software product. Such statement from the certification authorities enables the software provider to undergo the actual development of the software product.

The conceptual model defines concepts to capture the information pertaining to the software life cycle processes and related activities as described by the standard. The concepts were identified through the analysis of the standard. The supplements to DO-178C are irrelevant for the definition of the software life cycle activities as these documents do not contain additional guidelines regarding the definition of the software life cycle. The concepts that capture the definition of the software life cycle are depicted in Figure 3.1 and Figure 3.2. These concepts are described in the following subsections.

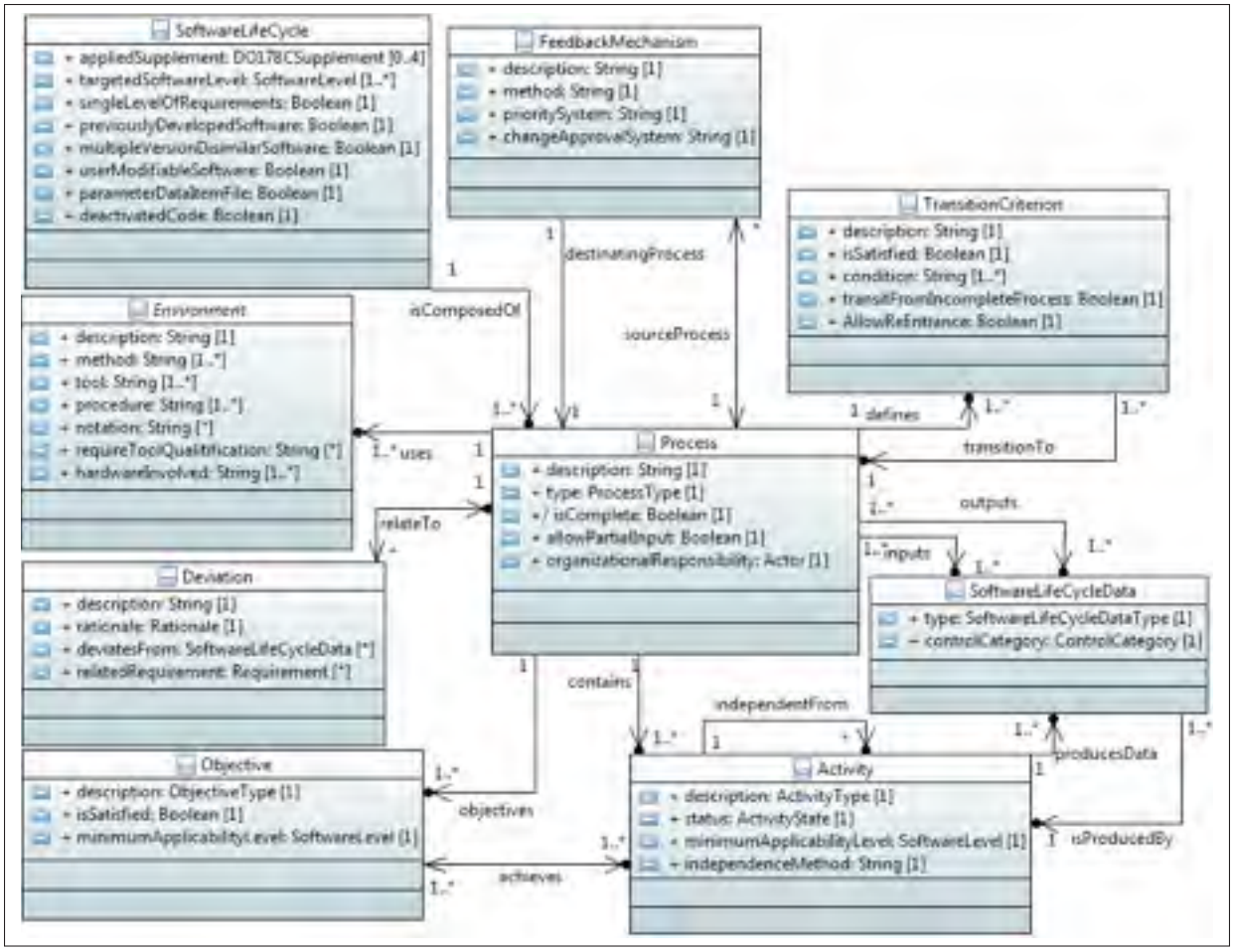


Figure 3.1 DO-178C software life cycle conceptual model.

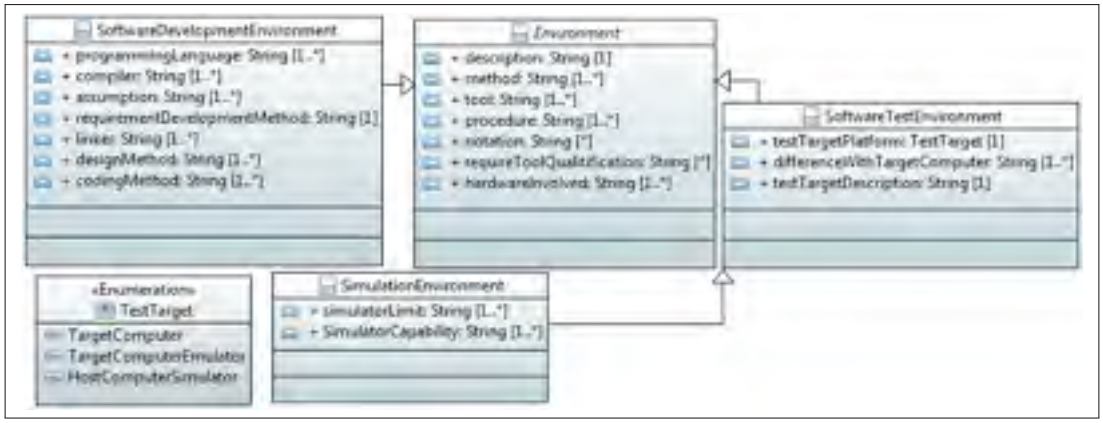


Figure 3.2 DO-178C software life cycle environment conceptual model.

3.2.1 Activity

Definition

The "Activity" concept identifies the tasks that must be carried out to meet an objective. These tasks are defined by DO-178C. The number of activities varies depending on the software level that a project targets.

Attributes

Name	Description
description	Offers the textual description of the activity. It also provides the reference to the chapter where the activity is defined in the standard. The following is an example of activity. 5.1.2.g - Derived high-level requirements and the reason for their existence should be defined.
status	The status of the activity. Examples of status include "In Progress", "Pending", "Terminated", "Under Review" and "Under Correction"
minimumApplicabilityLevel	Identifies the minimum software level for which the activity must be performed. As a result if the software level is set to D, the activity must be performed for software levels C, B and A.
independenceMethod	Specifies the method in use to provide independence from another activity when required. A mean to achieve independence is provided when another development team performs the activity from which independence is required.

Relationships

Name	Description
achieves	Identifies the Objectives (one or more) that are achieved by the activity.
independentFrom	Specific activities are required to be performed with independence from other activities. As such, this relationship identifies the activities from which an activity must be independent.
producesData	An activity produces data that are part of the evidences required for certification. The data produced by an activity may include one or more SoftwareLifeCycleData

Constraints

Constraint description	Software level	Introduced by supplement
if the *independentFrom* association is not empty, the activity shall provide a description of the method used to ensure that the activity is performed with independence.	Software level D	*None*

3.2.2 Deviation

Definition

The "Deviation" concept identifies a deviation that might occur from a plan, standard or requirement. Deviations are important as they must be submitted to the certification authorities.

Attributes

Name	Description
description	Describes what are the changes between the original plan, standard, or requirement and the result of the actual deviation. In other words it specifies the actions or decisions that resulted in a deviation.
rationale	Specifies the reasons why the deviation occurs.
deviatesFrom	Identifies one or more plan or standard from which the deviation occurs. "Software Verification Plan", "Software Requirements Standards"
relatedRequirement	Identifies zero or more requirements from which the deviation occurs. "HLR-1", "SRATS-1"

Relationships

Name	Description
relateTo	Identifies the process that is concerned by the deviation.

3.2.3 Environment

Definition

The "Environment" concept specifies the tools, procedures and notations that are used to perform the activities related to a process.

Attributes

Name	Description
description	The description of the environment.
method	Description of the methods related to the use of the specified environment.
tool	Identifies the tools used in the context of the specified environment. Example of tool includes IBM Rational DOORS for the specification of requirements.
procedure	Description of the procedures related to the environment.
notation	Specifies the notations used for the environment
requireToolQualification	Identifies the tools from the "tool" attribute that require to be qualified as defined by DO-330.
hardwareInvolved	Identifies the hardware involved for the specified environment.

3.2.4 FeedbackMechanism

Definition

The "FeedbackMechanism" concept provides a description of the way that feedback is provided by a process to another.

Attributes

Name	Description
description	Description of the feedback mechanism. Example of feedback includes the use of reports.
method	Description of the method used to communicate feedback between processes. Feedback may be communicated through the use of email or dedicated repository

Name	Description
prioritySystem	Textual description of the system used to assess and prioritize the created feedback.
changeApprovalSystem	Textual description of the system used to approve changes that are related to the created feedback.

Relationships

Name	Description
sourceProcess	Identifies the process that defines the feedback mechanism.
destinatingProcess	Identifies the process that receives feedback.

3.2.5 Objective

Definition

The "Objective" concept represents the requirements that should be met in order to demonstrate compliance with the standard (RTCA, 2011a).

Attributes

Name	Description
description	The description of the objective. "5.1.1.a - High-level requirements are developed."
isSatisfied	Specifies if the objective has been satisfied.
minimumApplicabilityLevel	The minimum software level applicable to the specified objective. Annexe I displays the objectives for software planning process and their software level.

Relationships

Name	Description
achieves	Identifies the activities, one or more, that achieve a specific objective.

3.2.6 Process

Definition

The "Process" concept represents a collection of activities performed in the software life cycle to produce various outputs or the software product (RTCA, 2011a) and to enable the achievement of a set of objectives.

Attributes

Name	Description
description	The description of the process. The following is the description of the software requirement process. "The high-level requirements are refined through one or more iterations in the software design process to develop the software architecture and the low-level requirements that can be used to implement source code."
type	The kind of process. Examples of processes include "Software Design Process" and "Integration Process".
/isComplete	Specifies if the process is completed. This is derived from the statuses of the activities of the process and its objectives.
allowPartialInput	Specifies if the process may begin its activities using incomplete input data.

Name	Description
organizationalResponsibility	Specifies the service, team, or person(s) responsible for the specified process. A software engineer may be responsible for the process.

Relationships

Name	Description
contains	Identifies the activities that are performed in the context of the specified process.
defines	Specifies the transition criterion for the process that are to be satisfied in order to transit to another process.
objectives	Identifies the objectives that are attached to the specified process.
uses	Specifies the environment(s) used to perform the activities of the specified process.
outputs	Specifies the list of software life cycle data that the process outputs.
inputs	Specifies the list of software life cycle data that the process receives as input.
sourceProcess	Identifies the feedback mechanism(s) defined by the specified process
relateTo	Identifies the deviation related to the specified process.

Constraints

- **Processes Inputs**

Constraint description	**Software level**	**Introduced by supplement**
If the process type value is set to ***SoftwareRequirementsProcess***, then the software life cycle data received as **inputs** by the process shall contain the following elements: *SystemRequirements*, *HardwareInterface*, *SystemArchitecture*, *Software Development Plan* and *Software Requirements Standards*. When DO-331 is used **inputs** shall also contain *Software Model Standards*.	Software level D	None
If the process type value is set to ***SoftwareDesignProcess***, then the software life cycle data received as **inputs** by the process shall contain the following elements: *SoftwareRequirementsData*, *Software Development Plan* and *Software Design Standards*. When DO-331 is used **inputs** shall also contain *Software Model Standards*.	Software level D	None
If the process type value is set to ***SoftwareCodingProcess***, then the software life cycle data received as **inputs** by the process shall contain the following elements: *Software Design Description*, *Software Development Plan* and *Software Code Standards*.	Software level D	None
If the process type value is set to ***IntegrationProcess***, then the software life cycle data received as **inputs** by the process shall contain the following elements: *Software Design Description* and *Source Code*.	Software level D	None

Constraint description	**Software level**	**Introduced by supplement**
If the process type value is set to ***SoftwareVerificationProcess***, then the software life cycle data received as **inputs** by the process shall contain the following elements: *SystemRequirements*, *SoftwareRequirementsData*, *SoftwareDesignDescription*, *TraceData*, *SourceCode*, *ExecutableObjectCode* and *SoftwareVerificationPlan*.	Software level D	None
If the process type value is set to ***CertificationLiaisonProcess***, then the software life cycle data received as **inputs** by the process shall contain the following elements: *Plan for Software Aspect of Certification*, *Software Accomplishment Summary* and *Software Configuration Index*.	Software level D	None

- **Processes Outputs**:

Constraint description	**Software level**	**Introduced by supplement**
If the process type value is set to ***SoftwarePlanningProcess***, then the software life cycle data received as **outputs** by the process shall contain the following elements: *Plan for Software Aspect of Certification*, *Software Development Plan*, *Software Verification Plan*, *Software Configuration Management Plan*, *Software Quality Assurance Plan*, *Software Requirements Standards*, *Software Design Standards*, *Software Code Standards* and *Software Verification Results*. When DO-331 is used **outputs** shall also contain *Software Model Standards*.	Software level D	None
If the process type value is set to ***SoftwareRequirementsProcess***, then the software life cycle data received as **outputs** by the process shall contain the following elements: *SoftwareRequirementsData* and *TraceData*.	Software level D	None
If the process type value is set to ***SoftwareDesignProcess***, then the software life cycle data received as **outputs** by the process shall contain the following elements: *SoftwareDesignDescription* and *TraceData*.	Software level D	None
If the process type value is set to ***SoftwareCodingProcess***, then the software life cycle data received as **outputs** by the process shall contain the following elements: *Source Code* and *Trace Data*.	Software level D	None

Constraint description	**Software level**	**Introduced by supplement**
If the process type value is set to ***IntegrationProcess***, then the software life cycle data received as **outputs** by the process shall contain the following elements: *Executable object code* and *Parameter Data item file*.	Software level D	None
If the process type value is set to ***SoftwareVerificationProcess***, then the software life cycle data received as **outputs** by the process shall contain the following elements: *SoftwareVerificationCasesAndProcedures*, *SoftwareVerificationResults* and *TraceData*.	Software level D	None
If the process type value is set to ***SoftwareConfigurationManagementProcess***, then the software life cycle data received as **outputs** by the process shall contain the following elements: *Software configuration management records*, *Software Configuration Index* and *Software Life Cycle Environment Configuration Index*.	Software level D	None
If the process type value is set to ***SoftwareQualityAssuranceProcess***, then the software life cycle data received as **outputs** by the process shall contain the following elements: *Software Quality Assurance Records*.	Software level D	None

Constraint description	Software level	Introduced by supplement
If the process type value is set to ***CertificationLiaisonProcess***, then the software life cycle data received as **outputs** by the process shall contain the following elements: *Plan for Software Aspect of Certification*, *Software Accomplishment Summary* and *Software Configuration Index*.	Software level D	None

- **Objectives**:

Constraint description	Software level	Introduced by supplement
When the process is of type ***SoftwarePlanningProcess***, the objectives of the process are the following: Objectives 4.1.b, 4.1.c, 4.1.e, 4.1.f and 4.1.g[1]	Software level C[2]	None
When the process is of type ***SoftwarePlanningProcess***, the objectives of the process are the following: Objectives 4.1.a, 4.1.d	Software level D	None
When the process is of type ***SoftwareRequirementProcess***, the objectives of the process are the following: Objectives 5.1.1.a and 5.1.1.b	Software level D	None
When the process is of type ***SoftwareRequirementProcess***, the objectives of the process are the following: Objective MB.5.1.1.c	Software level D	DO-331

[1] The way the objectives are referenced reuse the reference defined in DO-178C for each objective as observed in Appendix I.

[2] We remind the reader that this software level represents the minimum level to which the constraint applies. For instance, this constraint applies to software levels A, B and C.

Constraint description	**Software level**	**Introduced by supplement**
When the process is of type ***SoftwareDesignProcess***, the objectives of the process are the following: Objective 5.2.1.b	Software level C	None
When the process is of type ***SoftwareDesignProcess***, the objectives of the process are the following: Objectives 5.2.1.a	Software level D	None
When the process is of type ***SoftwareDesignProcess***, the objectives of the process are the following: Objective MB.5.2.1.c	Software level D	DO-331
When the process is of type ***SoftwareCodingProcess***, the objectives of the process are the following: Objective 5.3.1.a	Software level C	None
When the process is of type ***IntegrationProcess***, the objectives of the process are the following: Objective 5.4.1.a	Software level D	None
When the process is of type ***SoftwareVerificationProcess***, the objectives of the process are the following: Objective 6.4.4.c (modified condition/decision coverage and verification of additional code)[3].	Software level A	None
When the process is of type ***SoftwareVerificationProcess***, the objectives of the process are the following: Objectives 6.3.1.c, 6.3.2.c, 6.3.2.d, 6.3.3.c, 6.3.3.d, 6.3.4.c, 6.4.4.c (decision coverage)[4]	Software level B	None

[3] The achievement of this objective varies depending on the software level.

[4] The achievement of this objective varies depending on the software level.

Constraint description	Software level	Introduced by supplement
When the process is of type ***SoftwareVerificationProcess***, the objectives of the process are the following: Objectives 6.3.1.d, 6.3.1.e, 6.3.1.g, 6.3.2.a, 6.3.2.b, 6.3.2.e, 6.3.2.f, 6.3.2.g, 6.3.3.a, 6.3.3.b, 6.3.3.e, 6.3.4.a, 6.3.4.b, 6.3.4.d, 6.3.4.e, 6.3.4.f, 6.3.5.a, 6.6.b, 6.4.c, 6.4.d, 6.4.5.b, 6.4.5.c, 6.4.4.b, 6.4.4.c (statement coverage)[5], 6.4.4.d.	Software level C	None
When the process is of type ***SoftwareVerificationProcess***, the objectives of the process are the following: Objectives OO.6.7.1 and OO.6.8.1	Software level C	DO-332
When the process is of type ***SoftwareVerificationProcess***, the objectives of the process are the following: Objectives 6.3.1.a, 6.3.1.b, 6.3.1.f, 6.3.3.f, 6.6.a, 6.4.a, 6.4.b, 6.4.e, 6.4.4.a.	Software level D	None
When the process is of type ***SoftwareVerificationProcess***, the objectives of the process are the following: Objectives MB.6.8.3.2.a, MB.6.8.3.2.b and MB.6.8.3.2.c.	Software level D	DO-331
When the process is of type ***SoftwareConfigurationManagementProcess***, the objectives of the process are the following: Objectives 7.1.a, 7.1.b, 7.1.c, 7.1.d, 7.1.e, 7.1.f, 7.1.g, 7.1.h and 7.1.i	Software level D	None
When the process is of type ***SoftwareQualityAssuranceProcess***, the objectives of the process are the following: Objectives 8.1.a, 8.1.b and 8.1.c.	Software level C	None

[5] The achievement of this objective varies depending on the software level.

Constraint description	Software level	Introduced by supplement
When the process is of type ***SoftwareQualityAssuranceProcess***, the objectives of the process are the following: Objectives 8.1.b and 8.1.d.	Software level D	None
When the process is of type ***CertificationLiaisonProcess***, the objectives of the process are the following: Objectives 9.a, 9.b and 9.c.	Software level D	None

- **Activities**:

Constraint description	Software level	Introduced by supplement
When the process is of type ***SoftwarePlanningProcess*** the activities of the process are the following: Activities 4.3.b, 4.4.1, 4.4.2.a, 4.4.2.b, 4.4.2.c, 4.4.3, 4.2.b, 4.5, 4.3.a and 4.6.	Software level C	None
When the process is of type ***SoftwarePlanningProcess*** the activities of the process are the following: Activities 4.2.a, 4.2.c, 4.2.d, 4.2.e, 4.2.g, 4.2.i, 4.2.l, 4.3.c, 4.2.f, 4.2.h, 4.2.j and 4.2.k	Software level D	None
When the process is of type ***SoftwarePlanningProcess*** the activities of the process are the following: Activities MB.4.4.4.a, MB.4.4.4.b and MB.4.4.4.c	Software level C	DO-331
When the process is of type ***SoftwarePlanningProcess*** the activities of the process are the following: Activities MB.4.2.m, MB.4.2.n and MB.4.2.o	Software level D	DO-331

Constraint description	**Software level**	**Introduced by supplement**
When the process is of type ***SoftwareRequirementsProcess*** the activities of the process are the following: Activities 5.1.2.a, 5.1.2.b, 5.1.2.c, 5.1.2.d, 5.1.2.e, 5.1.2.f, 5.1.2.g, 5.1.2.h, 5.1.2.i, 5.1.2.j and 5.5.a	Software level D	None
When the process is of type ***SoftwareRequirementsProcess*** the activities of the process are the following: Activities MB.5.1.2.k, MB.5.1.2.l	Software level D	DO-331
When the process is of type ***SoftwareDesignProcess*** the activities of the process are the following: Activities 5.2.2.e, 5.2.2.f, 5.2.2.g, 5.2.3.a, 5.2.3.b, 5.2.4.a, 5.2.4.b, 5.2.4.c, 5.5.b, 5.2.2.b, 5.2.2.c.	Software level C	None
When the process is of type ***SoftwareDesignProcess*** the activities of the process are the following: Activities 5.2.2.a, 5.2.2.d.	Software level D	None
When the process is of type ***SoftwareDesignProcess*** the activities of the process are the following: Activity MB.5.2.2.h	Software level D	DO-331
When the process is of type ***SoftwareDesignProcess*** the activities of the process are the following: Activities OO.5.2.2.i and OO.5.5.d	Software level C	DO-332
When the process is of type ***SoftwareDesignProcess*** the activities of the process are the following: Activities OO.5.2.2.h, OO.5.2.2.i, OO.5.2.2.j, OO.5.2.2.k and OO.5.2.2.l.	Software level D	DO-332

Constraint description	**Software level**	**Introduced by supplement**
When the process is of type ***SoftwareCodingProcess*** the activities of the process are the following: Activities 5.3.2.a, 5.3.2.b, 5.3.2.c, 5.3.2.d and 5.5.c	Software level C	None
When the process is of type ***IntegrationProcess*** the activities of the process are the following: Activities 5.4.2.a, 5.4.2.b, 5.4.2.c, 5.4.2.d, 5.4.2.e and 5.4.2.f	Software level D	None
When the process is of type ***SoftwareConfigurationManagementProcess*** the activities of the process are the following: Activities 7.2.1.a, 7.2.1.b, 7.2.1.c, 7.2.1.d, 7.2.1.e, 7.2.2.a, 7.2.2.b, 7.2.2.c, 7.2.2.d, 7.2.2.e, 7.2.2.f, 7.2.2.g, 7.2.3.a, 7.2.3.b, 7.2.3.c, 7.2.4.a, 7.2.4.b, 7.2.4.c, 7.2.4.d, 7.2.4.e, 7.2.5.a, 7.2.5.b, 7.2.5.c, 7.2.5.d, 7.2.6.a, 7.2.6.b, 7.2.7.a, 7.2.7.b, 7.2.7.c, 7.2.7.d, 7.2.7.e, 7.4.a, 7.4.b, 7.5.a, 7.5.b and 7.5.c.	Software level D	None
When the process is of type ***SoftwareQualityAssuranceProcess*** the activities of the process are the following: Activities 8.2.b, 8.2.h, and 8.2.e	Software level C	None
When the process is of type ***SoftwareQualityAssuranceProcess*** the activities of the process are the following: Activities 8.2.a, 8.2.c, 8.2.d, 8.2.f, 8.2.h, 8.2.i, 8.2.g, 8.3.a, 8.3.b, 8.3.c, 8.3.d, 8.3.e, 8.3.f, 8.3.g, 8.3.h, and 8.3.i	Software level D	None

Constraint description	Software level	Introduced by supplement
When the process is of type ***CertificationLiaison-Process*** the activities of the process are the following: Activities 9.1.a, 9.1.b, 9.1.c, 9.2.a, 9.2.b, and 9.2.c	Software level D	None

3.2.7 SimulationEnvironment

Definition

The "SimulationEnvironment" concept provides the description of the environment used for simulation purpose.

Generalizations

Parent concept
SoftwareTestEnvironment (See 3.2.11)

Attributes

Name	Description
simulatorLimit	Describes the limitations that are imposed by the simulator.
simulatorCapability	Defines the capabilities of the simulator.

Constraints

Constraint description	Software level	Introduced by supplement
The use of a simulation environment shall comply with DO-331.	Software level D	*DO-331*

3.2.8 SoftwareDevelopmentEnvironment

Definition

The "SoftwareDevelopmentEnvironment" concept specifies various information related to the software development environment.

Generalizations

Parent concept
Environment (See 3.2.3)

Attributes

Name	Description
programmingLanguage	Identifies the programming languages used to define the source code of the software. Examples of programming languages include "ADA", "C++" and "C
compiler	Specifies the compilers that are used to produce the executable object code.
assumption	Identifies the assumptions that are made about the environment.

Name	Description
requirementDevelopmentMethod	Describes the method used for the development of the software requirements. "The Software Requirements Data will be structured and written based on the set of recommended practices on requirements engineering and management from the Requirements Engineering Management Handbook"
designMethod	Describes the method(s) used in order to specify the software design. "Some LLRs require to be expressed textually using design by contract."
codingMethod	Describes the method(s) used for the coding activities. "The LGCS is to be designed with UML 2.X and implemented in the Java programming language with the Java Development Kit (JDK) 8 and the Eclipse IDE. No special tools should be used to generate the code. Beyond the constraints stated here, no further constraints are placed on the use of support tools or hardware platforms."
linker	Identifies the linkers that are used to assemble the object code.

3.2.9 SoftwareLifeCycle

Definition

The "SoftwareLifeCycle" concept represents the ordered collection of processes that is considered sufficient and appropriate by an organization to produce a software product (RTCA,2011a). The software life cycle is defined by identifying the activities for each process, specifying a sequence for the processes (through transition criterion), and assigning responsibilities for the processes and as such for the activities.

Attributes

Name	Description
appliedSupplement	Identifies the supplements that are to be used for the developed software. They are additional guidelines that affect the evidence to be provided for certification when using specific software development technologies. Supplements include: DO-330, DO-331, DO-332 and DO-333.
targetedSoftwareLevel	Indicates the project's targeted software levels. Because a project might be composed of more than one component, each of them might be assigned different software levels. DO-178C defines 5 software levels: A, B, C, D and E
singleLevelOfRequirement	Specifies if the project uses a single level of requirement in order to specify its requirements. When this is set to false, the requirements are clearly organized into system requirements allocated to software (SRATs), high-level requirements and low-level requirements
previouslyDevelopedSoftware	Specifies if the software uses previously developed software.
multipleVersionDissimilarSoftware	Specifies if the software is a multiple version dissimilar software.
userModifiableSoftware	Specifies if the software is modifiable by its user.
paramaterDataItemFile	Specifies if the software uses parameter data item files.
deactivatedCode	Specifies if the source code contains deactivated code.

Relationships

Name	Description
isComposedOf	Identifies the processes that constitute the specified Software Life Cycle.

Constraints

Constraint description	Software level	Introduced by supplement
The value of *singleLevelOfRequirement* must be set to false when a model based technology is in use.	Software level D	DO-331

3.2.10 SoftwareLifeCycleData

Definition

The "SoftwareLifeCycleData" concept represents the documents that compile the data that have to be produced during the processes of the software life cycle. Such data are used to obtain certification of the software product and for post certification changes occurring to the software. Examples of produced data include requirements specification (SRD), requirements standards (SRS), plan for software aspects of certification (PSAC), software development plan (SDP), bug reports, test results (SVR), etc. DO-178C does not impose a specific form for the representation of these data.

Attributes

Name	Description
type	Specifies the kind of the data. Examples include "Software Requirements Data", "Plan for Software Aspects of Certification".
controlCategory	Identifies the control category placed upon the Software Life Cycle Data. Control categories influence the activities of the software configuration management process that applies to a SoftwareLifeCycleData. DO-178C defines two control categories: "Control Category 1", "Control Category 2".

Relationships

Name	Description
isProducedBy	Identifies the activity that produce data for the specified SoftwareLifeCycleData.

3.2.11 SoftwareTestEnvironment

Definition

The "SoftwareTestEnvironment" concept specifies various informations related to the used testing environments.

Generalizations

Parent concept
Environment (See 3.2.3)

Attributes

Name	Description
differenceWithTargetComputer	Describes of the differences between the target computer and the emulator or simulator used to perform the testing activities.
testTargetPlatform	Identifies the platform that is used to perform the testing activities. "TargetComputer", "TargetComputerEmulator", "HostComputerSimulator"
testTargetDescription	Description of the test platform used to perform the testing activities.

3.2.12 TransitionCriterion

Definition

The "TransitionCriterion" represents the minimum conditions defined by the software planning process to be satisfied in order to enter a process (RTCA,2011a).

Attributes

Name	Description
description	The textual description of the transition criteria. "The software verification process review have been performed."
isSatisfied	States whether the transition criteria is being met in order to enter its destination process.
condition	Specifies the conditions that constitute the transition criterion. "High-level requirements must be specified and reviewed"
transitFromIncompleteProcess	States whether a transition criteria allows transition from an incomplete process.
allowReEntrance	Specifies if the destination process can be re-entered.

Relationships

Name	Description
transitionTo	Specifies the process that shall be entered once the conditions of the transition criteria are satisfied.

3.3 Software Requirements Process

Requirements are used to specify the capabilities, conditions and limitations that a system must satisfy (Langer & Tautschnig, 2008). Requirements are usually classified into several categories. In a DO-178C context, requirements categories include: functional requirements, operational requirements, interface requirements, performance requirements, security requirements, maintenance requirements, certification requirements, safety related requirements, and other types of requirements. In the avionics context, the software requirements are developed from the system requirements allocated to software (SRATs). In fact, SRATs are successively developed into high-level requirements and low-level requirements.

The DO-178C profile intends to provide modeling constructs that enable the specification of requirements and their traceability. The guidelines related to the specification of requirements are provided as part of the software requirement process (specification of HLRs) and the software design process (specification of LLRs). However, the software design process does not focus solely on the definition of the LLRs. It also includes the definition of the software architecture. Because the concepts of HLRs and LLRs share many common properties, we have decided to define the LLRs concept within the conceptual model built from the analysis of the software requirement process. This analysis has been completed with the portion of the software design process pertaining to LLRs specification.

The concepts that capture the definition of requirements and their traceability as defined by DO-178C are depicted in Figure 3.3. These concepts are described in the following subsections.

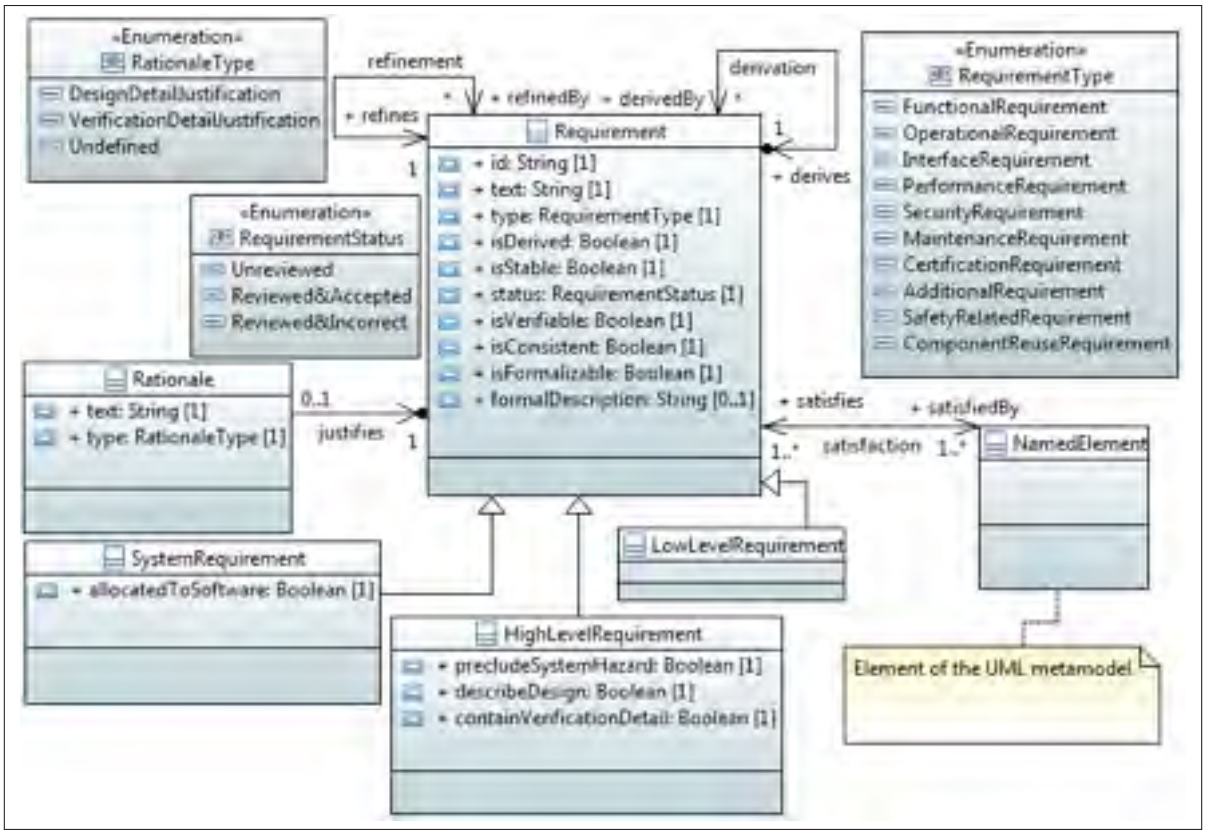


Figure 3.3 DO-178C requirements conceptual model.

3.3.1 HighLevelRequirement

Definition

The "HighLevelRequirement" (HLR) concept describes software requirements that are developed from the analysis of the system requirements that are allocated to software, safety-related requirements and the system architecture (RTCA, 2011a).

Generalizations

Parent concept
Requirement (See 3.3.4)

Attributes

Name	Description
precludeSystemHazard	Specifies if the HLR specification intends to prevent one or more of the identified system hazards.
describeDesign	Specifies if a HLR include design details. (As described by DO-178C, HLRs should not include such details.)
containVerificationDetail	Specifies if a HLR include verification details. (As described by DO-178C, HLRs should not include such details.)

Constraints

Constraint description	Software level	Introduced by supplement
Derived High-level Requirements must be justified by a Rationale.	Software level D	None
A high-level requirement must be traceable to the system requirement allocated to software it refines.	Software level D	None
When the *describeDesign* attribute is set to true, the reason for the description of design details should be justified by a rationale.	Software level D	None
When the *containVerificationDetail* attribute is set to true, a rationale should be provided to justify the capture of verification details in the requirement.	Software level D	None

3.3.2 LowLevelRequirement

Definition

The "LowLevelRequirements" (LLR) concept describes software requirements that are developed from the high-level requirements, derived requirements, and design constraints. Low-level requirements are requirements from which source code can be directly implemented without further information (RTCA, 2011a).

Generalizations

Parent concept
Requirement (See 3.3.4)

Constraints

Constraint description	**Software level**	**Introduced by supplement**
A derived low-level requirement must be justified by a rationale.	Software level C	None

3.3.3 Rationale

Definition

The "Rationale" concept purpose is to provide justification for the decisions that are made during the software life cycle.

Attributes

Name	Description
type	Identifies the type of rationale. Possible values include: "Undefined", "DesignDetailJustification" and "Verification-DetailJustification".
text	The rationale's text.

Constraints

Constraint description	Software level	Introduced by supplement
A rationale text must always be specified.	Software level D	None

3.3.4 Requirement

Definition

The "Requirement" concept is a general concept that describes all requirements including SRATs, HLRs and LLrs. A requirement's purpose is to describe what is to be performed by the system, or the software given a set of inputs and constraints (RTCA, 2011a). Requirements can be either functional or non-functional (i.e. specified by its type attribute). Requirements are traceable to higher or lower level of requirements through the refinement relationship. They are also traceable to derived requirements through the derivation relationship.

Attributes

Name	Description
id	The unique ID that is used to identify a requirement. Examples of ID include: "SRATS-10", "HLR-1" or "LLR-27".

Name	Description
text	The specification of the requirement. The following is an example of a requirement text. "If the validateSensorData function is active, the three readings are valid and have equal values, the overall sensor value shall be this common value and be valid."
type	Identifies the type of requirement. Possible values include: "FunctionalRequirement", "OperationalRequirement", "InterfaceRequirement", "PerformanceRequirement", "SecurityRequirement", "MaintenanceRequirement", "CertificationRequirement", "AdditionalRequirement", "SafetyRelatedRequirement" and "ComponentResuseRequirement".
isDerived	Identifies if a requirement is a derived requirement. Derived requirements are requirements that are not directly traceable to higher level requirements and/or are requirements that specify behavior beyond that specified by the system requirement or the high-level requirements.
isStable	Specifies if a requirement is stable. A stable requirement indicates that its specification shall not evolve.
status	Specifies the status of the verification activities related to the requirement. Possible values are: "Unreviewed", "Reviewed&Incorrect", "Reviewed&Accepted".
isVerifiable	Specifies if verification activities can be performed on the requirement.
isConsistent	Specifies if the requirement is consistent as defined by the review made of the requirement.

Name	Description
isFormalizable	Specifies if the requirement can be specified using a formal notation. If set to "True" this enables its representation and verification with a formal method.
formalDescription	A formal specification of the requirement.

Relationships

Name	Description
refinement	Specifies the refinement of a requirement into a lower level of requirement, enabling the bi-directional traceability of the involved requirements. The relationship reads as follows: • **refinedBy**: a requirement is refined by zero or more requirements. • **refines**: a requirement refines another.
derivation	Identifies the specification of derived requirements, enabling bi-directional traceability between the derived requirement and the requirement being derived. The relationship reads as follows: • **derivedBy**: a requirement is derived by zero or more requirements. • **derives**: a requirement derives another.
satisfaction	Identifies the design elements that satisfy a requirement. The relationship reads as follows: • **satisfiedBy**: a requirement is satisfied by one or more design elements. • **satisfies**: a design element satisfies one or more requirements.

Constraints

Constraint description	Software level	Introduced by supplement
When the *isFormalizable* attribute is set to true, the attribute *formalDescription* shall provide a formal specification for the requirement.	Software level D	DO-333
A requirement *id* must be unique	Software level D	None
A requirement *id* must be specified.	Software level D	None
A requirement *text* attribute must not be empty.	Software level D	None

3.3.5 SystemRequirement

Definition

The "SystemRequirement" (SRAT) concept represents requirements that describe at the system level the functionality that the system, as a whole, must fulfill in order to satisfy the stakeholders needs.

Generalizations

Parent concept
Requirement (See 3.3.4)

Attributes

Name	Description
allocatedToSoftware	Identifies if the system requirement is allocated to software.

3.4 Software Verification Process

The verification activities defined by the software verification process of DO-178C involve the technical assessment of the output of the software planning process, the software development process, the software coding process, the integration process and the software verification process itself. The subset of the conceptual model related to the software verification process defines the concepts that capture the information related to the activities that have to be performed as part of the software verification process as required by DO-178C.

The main purpose of these concepts is to capture the information generated during reviews, analyses and testing activities that are performed in order to demonstrate the software ability to execute safely in its operating environment and that the developed software product complies with its associated airworthiness requirements.

The concepts that we have defined in order to capture these information and their related traceability are introduced in Figure 3.4 These concepts are described in the following subsections.

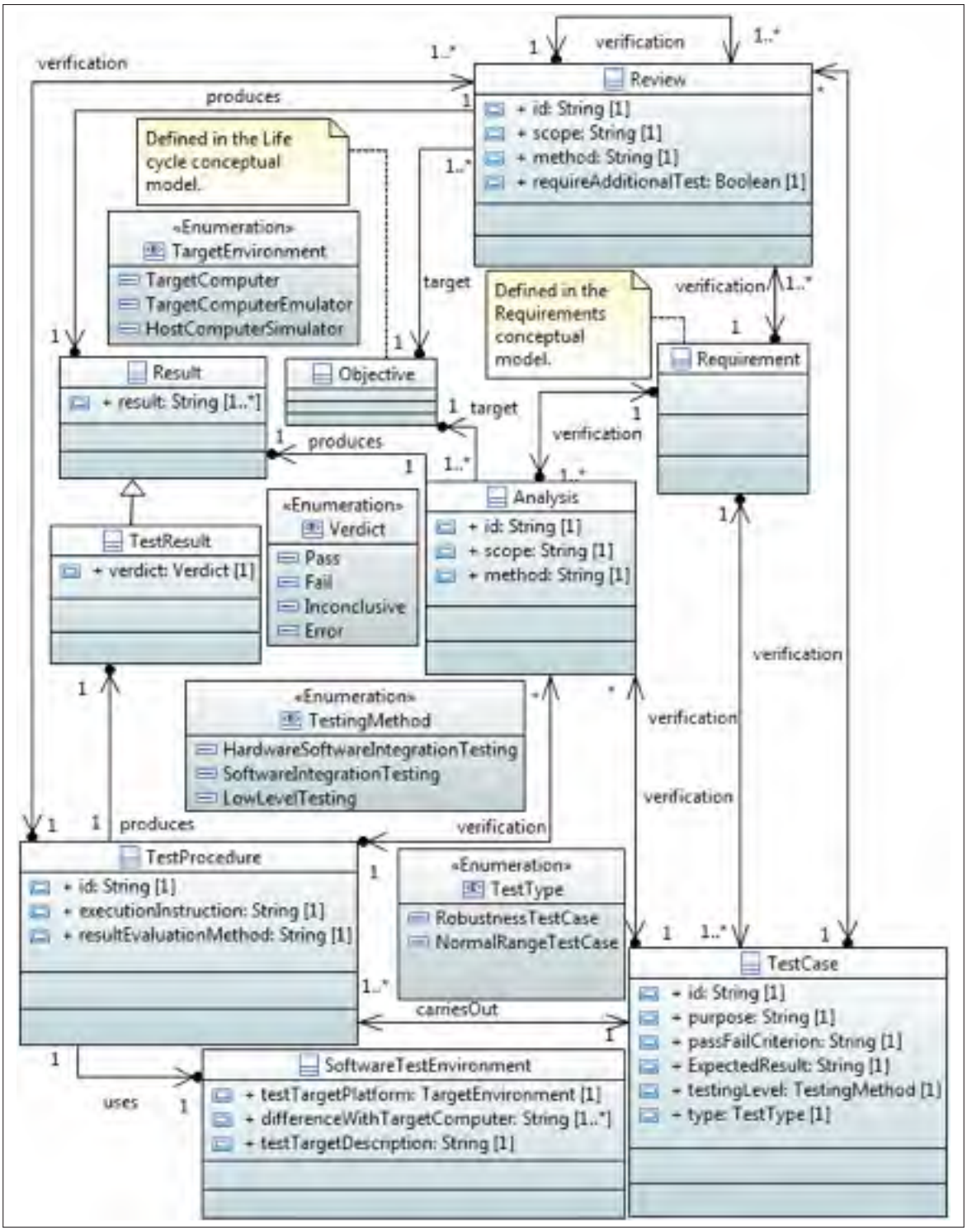


Figure 3.4 Verification conceptual model.

3.4.1 Analysis

Definition

The "Analysis" concept is used to define an analysis that could be carried out on a requirement, a test case or a review. Analyses provide repeatable evidences of correctness for the element(s) under scrutiny.

Attributes

Name	Description
id	An unique ID used to identify an analysis. Example of ID includes: "Analysis 1".
scope	Description of the scope of the analysis.
method	The method that is used to perform the analysis

Relationships

Name	Description
produces	Specifies the result of the performed analysis.
target	Identifies the objective that the analysis intends to meet.
verification	Identifies the requirement on which the analysis is performed.
verification	Identifies the test case on which the analysis is performed.
verification	Identifies the test procedure on which the analysis is performed.

3.4.2 Result

Definition

The "Result" concept captures the information resulting from an analysis or a review.

Attributes

Name	Description
result	Description of the results obtained from an analysis or a review.

3.4.3 Review

Definition

The "Review" concept is used to define a review. Reviews provide a qualitative assessment of correctness for the element(s) under scrutiny. Requirements, test cases, test procedures, analyses and reviews can be reviewed.

Attributes

Name	Description
id	The unique ID used to identify the review. Example of ID includes: "Review 1".
scope	Description of the scope of the review.
method	The method used to perform the review. A review may be a checklist.

Name	Description
requireAdditionalTest	Specifies if the review needs additional tests in order to be carried out and as such, to express the results of the performed review.

Relationships

Name	Description
produces	Specifies the result of the performed review.
target	Identifies the objective that the review intends to meet.
verification	Identifies the requirement on which the review is performed.
verification	Identifies the test case on which the review is performed.
verification	Identifies the test procedure on which the review is performed.
verification	Identifies the review on which a review is performed.

3.4.4 TestCase

Definition

The "TestCase" concept represents the set of inputs, execution conditions and expected results developed for a particular testing objective. Examples of testing objectives include the execution of a specific program path or the verification of compliance against a specific requirement.

Attributes

Name	Description
id	The unique ID used to identify the test case. Example of ID includes: "TestCase 1".
purpose	Description of the objective of test case.
passFailCriterion	Criterion that defines the conditions to be met by the test case after its execution in order to define if it passes or fails.
expectedResult	Description of the expected results for the test case.
testingLevel	The kind of test that is specified. DO-178C testing levels include: "HardwareSoftwareIntegrat-ionTesting", "SoftwareIntegrationTesting", "LowLevelTesting".
type	Specifies the type of test case. Possible values are: "RobustnessTestCase" and "NormalRangeTestCase".

Relationships

Name	Description
carriesOut	Identifies the test procedures that describe how to execute the specified test case.
verification	Identifies the requirement that is verified by the specified test case.
verification	Identifies the reviews that perform a review of the specified test case.
verification	Identifies the analysis that perform an analysis on the specified test case.

3.4.5 TestProcedure

Definition

The "TestProcedure" concept captures the detailed instructions for the set-up and execution of a given test case, along with the instructions required for the evaluation of the related test case execution results.

Attributes

Name	Description
id	The unique ID used to identify the procedure. Example of ID includes: "test procedure 1".
executionInstruction	Description of the instructions required to execute the related test case.
resultEvaluationMethod	The method used to analyze the result obtained from performing the test.

Relationships

Name	Description
carriesOut	Identifies the test case that is related to the specified test procedure.
produces	Specifies the result of the performed test.
uses	Specifies the environments that are used in order to perform the test.
verification	Identifies the reviews that perform a review of the specified test procedure.
verification	Identifies the analyses that perform an analysis on the specified test procedure.

3.4.6 TestResult

Definition

The "TestResult" concept captures the resulting information related to a test case execution. The test results are meant to be compared with the expected test results provided by the test case in order to evaluate the results of the execution of the related test case.

Generalizations

Parent concept
Result (See 3.4.2)

Attributes

Name	**Description**
verdict	Specifies the final result of the performed test. The value of the verdict may be: "None", "Pass", "Inconclusive", "Fail" or "Error".

CHAPTER 4

AN ASSURANCE LEVEL SENSITIVE UML PROFILE FOR SUPPORTING DO-178C

In order to describe the proposed profile, this chapter is organized as follows. In Section 4.1, we present the architecture of the profile. In Section 4.2, the template used to describe the stereotypes of the proposed profile is introduced. In Section 4.3, the stereotypes of the LifeCycle package are introduced. in Section 4.4, the stereotypes of the Requirements package are introduced. Finally, in Section 4.5 the stereotypes of the Verification package are introduced.

4.1 Profile architecture

Based on the conceptual model presented in Chapter 3, we built an UML profile that supports DO-178C. In particular the profile enables capturing the information related to the planning, requirement specification and verification processes. Our profile has the unique characteristic of being sensitive to the assurance level of the software and specifying the information related to the software life cycle.

In order to preserve the logical structure defined in DO-178C, the proposed profile was organized into different packages. Figure 4.1 depicts the packages of the profile and their inter-relationships. The content of these packages resulted from the mapping of the conceptual model presented in Chapter 3 to the UML metamdel. These packages are the following:

- **LifeCycle**: This package groups the concepts that pertain to the definition of a project life cycle as prescribed by the standard.
- **Requirements**: This package includes the concepts that are related to requirements definition and management as defined by the standard.
- **Verification**: This package groups the concepts that pertain to validation and verification activities as defined by the standard.

- **Types**[1]: This package defines a set of types used by other packages of the profile.

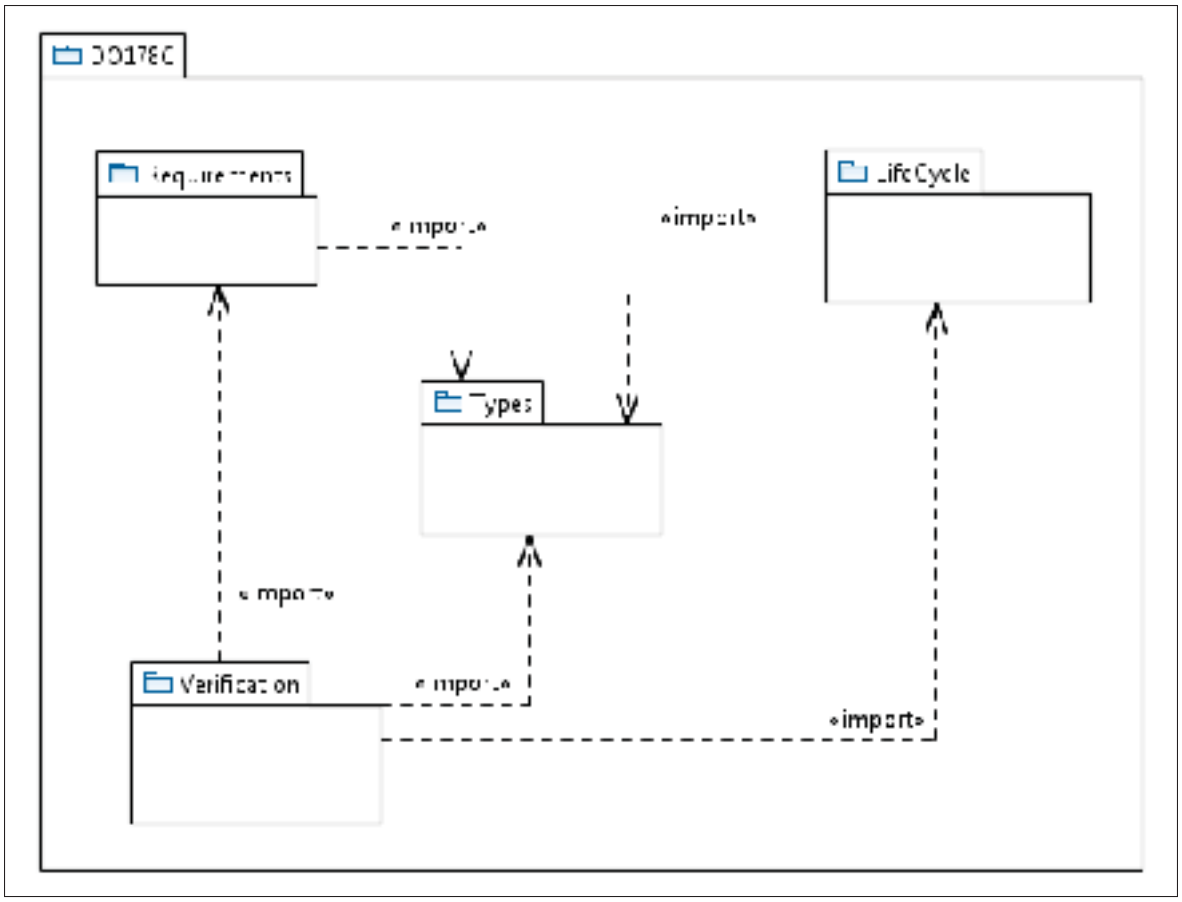


Figure 4.1 The package structure of the proposed profile.

4.2 UML Profile - Template description

We have developed a template with the goal of providing the reader with a clear and easy to understand specification for the stereotypes of the introduced profile. In order to build this template, we have explored the existing relevant literature in order to gain knowledge on the current practices used to describe UML profiles. This exploration led us to the following documents:

[1] The content of the Types package is provided in Appendix I.

- the UML specification (OMG, 2015)
- Meta Object Facility (MOF) Core Specification (OMG, 2015)
- the System Modeling Language (SysML) specification (OMG, 2014)
- Modeling and Analysis of Real-Time Embedded Systems (MARTE) specification (OMG, 2011)
- the UML testing profile (UTP) (OMG, 2014)
- the UML Profile for BPMN Processes (OMG, 2014)
- the UML Profile for Advanced and Integrated Telecommunication Services (TelcoML) (OMG, 2014)

Our template has been built upon a combination of elements that were used in the aforementioned profile specifications. We believe using a template that follows a similar structure to the ones used by standardized profiles will help the reader in the comprehension of the content of our profile.

The template to describe the stereotypes of our profile is composed of the following sections:

Description

Provides a general description of the specified stereotype.

Related concept

Identifies the concept of the conceptual model that the stereotype represents.

Extensions [2]

Provides the list of the UML metaclasses being extended by the specified stereotype.

This section is provided using a table as follows:

Base Metaclass	**Explanation**
Metaclass name	*Explanation*

Generalizations [2]

Provides the list of stereotypes that are specialized by the the specified stereotype.

This section is provided using a table as follows:

Parent class name	**Explanation**
Parent class name	*Explanation*

Attributes

The attribute section of the template provides the list of attributes for the stereotype being specified.

The list of attributes is provided using a table as follows:

[2] Only one of the sections "Extensions" or "Generalizations" is to be used for the description of a stereotype as a stereotype can either extend a metaclass of the UML metamodel or generalize another stereotype.

Name	Type	Multiplicity	Description	Is derived
Attribute name	*type*	*[0..*]*	*References the description of the related attribute in the conceptual model*	*True or false*

Associations

Provides the list of associations for the specified stereotype. It is provided using a table as follows:

Name	Type	Multiplicity	Description	Opposed member end
Association name	*type*	*[0..*]*	*Description*	*Opposed member end name*

Constraints

Provides the list of constraints that applies to the specified stereotype. To avoid redundancy, in this section we refer to the constraints already introduced in Chapter 3 that are related to the concept represented to by the specified stereotype.

4.3 LifeCycle Package

Figure 4.2 and Figure 4.3 introduce the mapping of the conceptual models provided respectively in Figure 3.1 and Figure 3.2 to the UML meta-model.

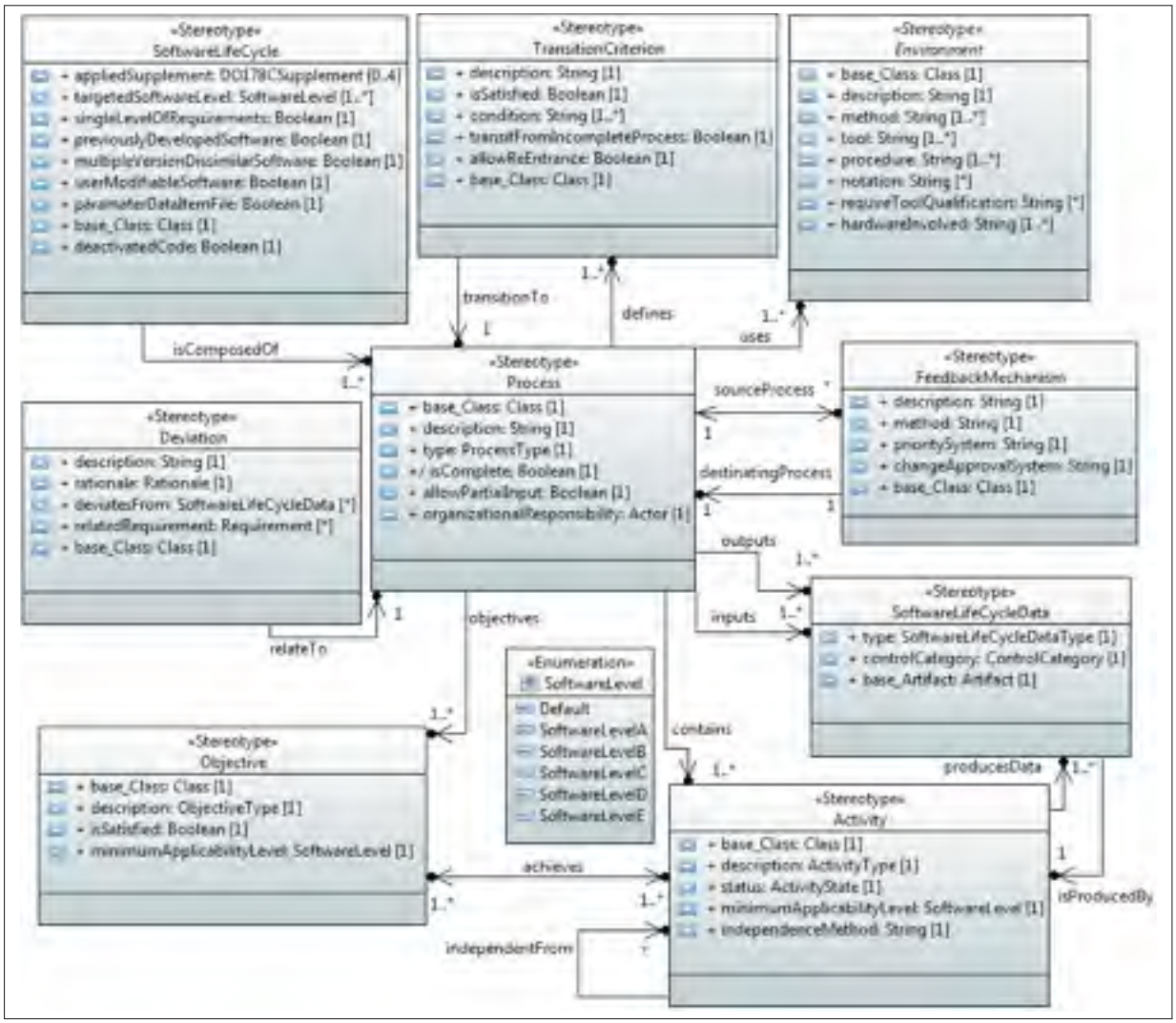


Figure 4.2 LifeCycle package diagram.

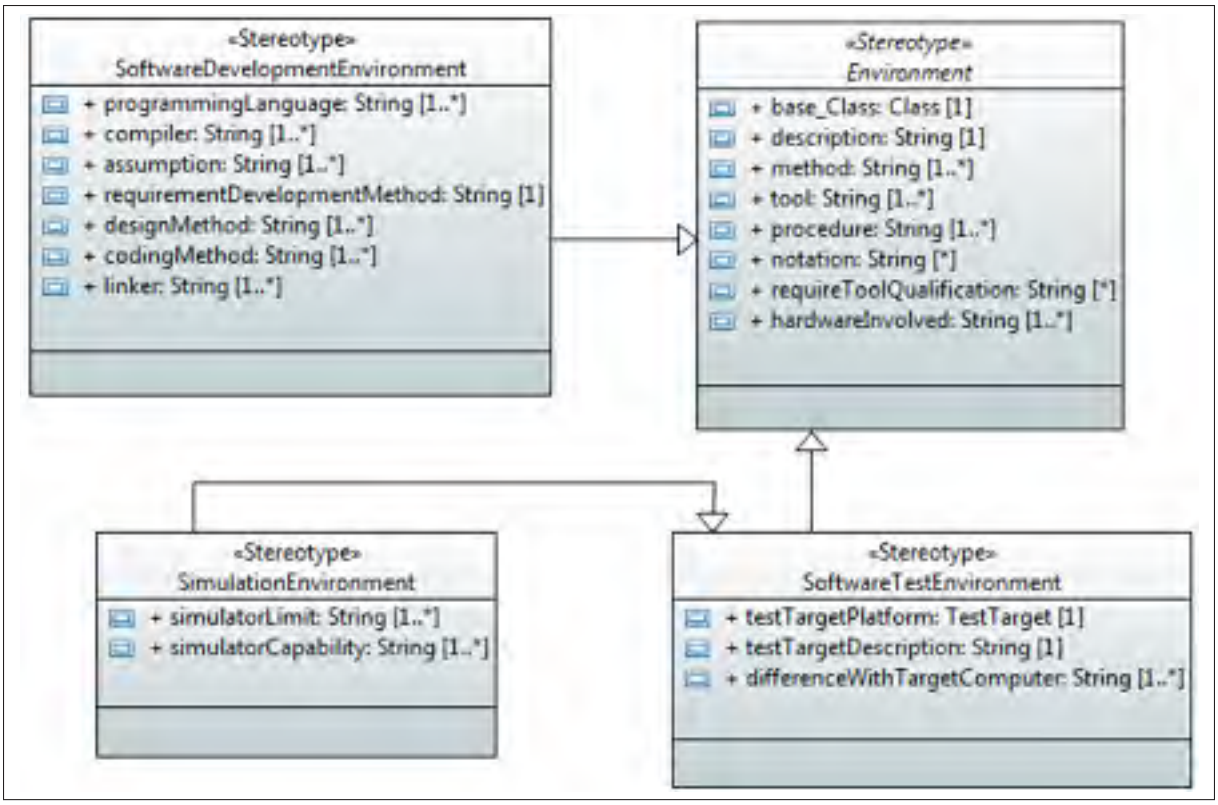


Figure 4.3 LifeCycle package diagram, Environment related entities.

The stereotypes of the LifeCycle Package are the following:

4.3.1 *«Activity»*

Description

The «Activity» stereotype represents the tasks that must be carried out to meet an objective. These tasks are defined by DO-178C. The number of activities varies depending on the software level that a project targets.

Related concept

See *Activity* (3.2.1).

Extensions

Base Metaclass	Explanation
Class	The metaclass Class allows us to represent any object without capturing unnecessary semantic of the UML metamodel that is not relevant for the introduced stereotype.

Attributes

Name	Type	Multiplicity	Description	Is derived
description	ActivityType	[1]	See *description* from Activity (3.2.1).	False
status	ActivityState	[1]	See *status* from Activity (3.2.1).	False
minimumApplicabilityLevel	SoftwareLevel	[1]	See *minimumApplicabilityLevel* from Activity (3.2.1).	False
independenceMethod	String	[1]	See *independenceMethod* from Activity (3.2.1).	False

Associations

Name	Type	Multiplicity	Description	Opposed member end
achieves	Association	[1..*]	See *achieves* from Activity (3.2.1).	«Objective»
independentFrom	Association	[*]	See *independentFrom* from Activity (3.2.1).	«Activity»
producesData	Association	[1..*]	See *procudesData* from Activity (3.2.1).	«SoftwareLifeCycleData»

Constraints

See *Activity* (3.2.1).

4.3.2 *«Deviation»*

Description

The «Deviation» stereotype represents a deviation that might occur from a plan, standard or requirement. Deviation are important as they must be submitted to the certification authorities.

Related concept

See *Deviation* (3.2.2).

Extensions

Base Metaclass	Explanation
Class	The metaclass Class allows us to represent any object without capturing unnecessary semantic of the UML metamodel that is not relevant for the introduced stereotype.

Attributes

Name	Type	Multiplicity	Description	Is derived
description	String	[1]	See *description* from Deviation (3.2.2).	False
rationale	«Rationale»	[1]	See *rationale* from Deviation (3.2.2).	False
deviatesFrom	«SoftwareLifeCycleData»	[*]	See *deviatesFrom* from Deviation (3.2.2).	False
relatedRequirement	«Requirements»	[*]	See *relatedRequirement* from Deviation (3.2.2).	False

Associations

Name	Type	Multiplicity	Description	Opposed member end
relateTo	Association	[1]	See *relateTo* from Deviation (3.2.2).	«Process»

4.3.3 *«Environment»*

Description

«Environment» is an abstract stereotype that specifies the tools, procedures and notations that are used to perform the activities related to a process.

Related concept

See *Environment* (3.2.3).

Extensions

Base Metaclass	Explanation
Class	The metaclass Class allows us to represent any object without capturing unnecessary semantic of the UML metamodel that is not relevant for the introduced stereotype.

Attributes

Name	Type	Multiplicity	Description	Is derived
description	String	[1]	See *description* from Environment (3.2.3).	False
method	String	[1]	See *method* from Environment (3.2.3).	False
tool	String	[1..*]	See *tool* from Environment (3.2.3).	False
procedure	String	[1..*]	See *procedure* from Environment (3.2.3).	False
notation	String	[*]	See *notation* from Environment (3.2.3).	False
requireToolQualification	String	[*]	See *requireToolQualification* from Environment (3.2.3).	False
hardwareInvolved	String	[1..*]	See *hardwareInvolved* from Environment (3.2.3).	False

4.3.4 *«FeedbackMechanism»*

Description

The «FeedbackMechanism» stereotype enable the specification of the way that feedback is provided by a process of the software life cycle to another.

Related concept

See *FeedbackMechanism* (3.2.4).

Extensions

Base Metaclass	Explanation
Class	The metaclass Class allows us to represent any object without capturing unnecessary semantic of the UML metamodel that is not relevant for the introduced stereotype.

Attributes

Name	Type	Multiplicity	Description	Is derived
description	String	[1]	See *description* from FeedbackMechanism (3.2.4).	False
method	String	[1]	See *method* from FeedbackMechanism (3.2.4).	False

Name	Type	Multiplicity	Description	Is derived
prioritySystem	String	[1]	See *prioritySystem* from FeedbackMechanism (3.2.4).	False
changeApprovalSystem	String	[1]	See *changeApprovalSystem* from FeedbackMechanism (3.2.4).	False

Associations

Name	Type	Multiplicity	Description	Opposed member end
destinatingProcess	Association	[1]	See from FeedbackMechanism (3.2.4).	«Process»
sourceProcess	Association	[1]	See *sourceProcess* from FeedbackMechanism (3.2.4).	«Process»

4.3.5 *«Objective»*

Description

The «Objective» stereotype represents the requirements that should be met in order to demonstrate compliance with the standard (RTCA, 2011a).

Related concept

See *Objective* (3.2.5).

Extensions

Base Metaclass	Explanation
Class	The metaclass Class allows us to represent any object without capturing unnecessary semantic of the UML metamodel that is not relevant for the introduced stereotype.

Attributes

Name	Type	Multiplicity	Description	Is derived
description	ObjectiveType	[1]	See *description* from Objective (3.2.5).	False
isSatisfied	Boolean	[1]	See *isStatisfied* from Objective (3.2.5).	False
minimumApplicabilityLevel	SoftwareLevel	[1]	See *minimumApplicabilityLevel* from Objective (3.2.5).	False

Associations

Name	Type	Multiplicity	Description	Opposed member end
achieves	Association	[1..*]	See *achieves* from Objective (3.2.5).	«Activity»

4.3.6 *«Process»*

Description

The «Process» stereotype represents a collection of activities performed in the software life cycle to produce various outputs or the software product (RTCA, 2011a) and to enable the achievement of a set of objectives.

Related concept

See *Process* (3.2.6).

Extensions

Base Metaclass	Explanation
Class	The metaclass Class allows us to represent any object without capturing unnecessary semantic of the UML metamodel that is not relevant for the introduced stereotype.

Attributes

Name	Type	Multiplicity	Description	Is derived
description	String	[1]	See *description* from Process (3.2.6).	False
type	ProcessType	[1]	See *type* from Process (3.2.6).	False
/isComplete	Boolean	[1]	See *isComplete* from Process (3.2.6).	True
allowPartialInput	Boolean	[1]	See *allowPartialInput* from Process (3.2.6).	False
organizationalResponsibility	Actor	[1]	See *organizationalResponsibility* from Process (3.2.6).	False

Associations

Name	Type	Multiplicity	Description	Opposed member end
contains	Association	[1..*]	See *performs* from Process (3.2.6).	«Activity»
sourceProcess	Association	[*]	See *sourceProcess* from Process (3.2.6).	«FeedbackMechanism»
defines	Association	[1..*]	See *defines* from Process (3.2.6).	«TransitionCriterion»

Name	Type	Multiplicity	Description	Opposed member end
objectives	Association	[1..*]	See *objectives* from Process (3.2.6).	«Objective»
uses	Association	[1..*]	See *uses* from Process (3.2.6).	«Environment»
outputs	Association	[1..*]	See *outputs* from Process (3.2.6).	«SoftwareLifeCycleData»
inputs	Association	[1..*]	See *inputs* from Process (3.2.6).	«SoftwareLifeCycleData»
relateTo	Association	[*]	See *relateTo* from Process (3.2.6).	«Deviation»

Constraints

See *Process* (3.2.6).

4.3.7 *«SimulationEnvironment»*

Description

The «SimulationEnvironment» stereotype describes the environment used for simulation purpose.

Related concept

See *SimulationEnvironment* (3.2.7).

Generalizations

Parent class name	Explanation
«SoftwareTestEnvironment»	A simulation environment is a specialization of the «Environment» stereotype.

Attributes

Name	Type	Multiplicity	Description	Is derived
simulatorLimit	String	[1..*]	See *simulatorLimit* from SimulationEnvironment (3.2.7).	False
simulatorCapability	String	[1..*]	See *simulatorCapability* from SimulationEnvironment (3.2.7).	False

Constraints

See *SimulationEnvironment* (3.2.7).

4.3.8 *«SoftwareDevelopmentEnvironment»*

Description

The «SoftwareDevelopmentEnvironment» stereotype specifies various information related to the software development environment.

Related concept

See *SoftwareDevelopmentEnvironment* (3.2.8).

Generalizations

Parent class name	Explanation
«Environment»	A software development environment is a specialization of the «Environment» stereotype.

Attributes

Name	Type	Multiplicity	Description	Is derived
programmingLanguage	String	[1..*]	See *programmingLanguage* from SoftwareDevelopmentEnvironment (3.2.8).	False
compiler	String	[1..*]	See *complier* from SoftwareDevelopmentEnvironment (3.2.8).	False
assumption	String	[1..*]	See *assumption* from SoftwareDevelopmentEnvironment (3.2.8).	False
requirementDevelopmentMethod	String	[1]	See *requirementDevelopmentMethod* from SoftwareDevelopmentEnvironment (3.2.8).	False

Name	Type	Multiplicity	Description	Is derived
designMethod	String	[1..*]	See *designMethod* from SoftwareDevelopmentEnvironment (3.2.8).	False
codingMethod	String	[1..*]	See *codingMethod* from SoftwareDevelopmentEnvironment (3.2.8).	False
linker	String	[1..*]	See *linker* from SoftwareDevelopmentEnvironment (3.2.8).	False

4.3.9 *«SoftwareLifeCycle»*

Description

The «SoftwareLifeCycle» stereotype represents the ordered collection of processes that is considered sufficient and appropriate by an organization to produce a software product (RTCA,2011a). The software life cycle is defined by identifying the activities for each process, specifying a sequence for the processes (through transition criterion) and assigning responsibilities for the processes and as such for the activities.

Related concept

See *SoftwareLifeCycle* (3.2.9).

Extensions

Base Metaclass	Explanation
Class	The metaclass Class allows us to represent any object without capturing unnecessary semantic of the UML metamodel that is not relevant for the introduced stereotype.

Attributes

Name	Type	Multiplicity	Description	Is derived
appliedSupplement	DO178CSupplement	[0..4]	See *appliedSupplement* from SoftwareLifeCycle (3.2.9).	False
targetedSoftwareLevel	SoftwareLevel	[1..*]	See *targetedSoftwarelevel* from SoftwareLifeCycle (3.2.9).	False
singleLevelOfRequirements	Boolean	[1]	See *singleLevelOfRequirement* from SoftwareLifeCycle (3.2.9).	False

Name	Type	Multiplicity	Description	Is derived
previouslyDeveloped- Software	Boolean	[1]	See *previouslyDevelopedSoftware* from SoftwareLifeCycle (3.2.9).	False
multipleVersionDissimilarSoftware	Boolean	[1]	See *multipleVersionDissimilarSoftware* from SoftwareLifeCycle (3.2.9).	False
userModifiableSoftware	Boolean	[1]	See *userModifiableSoftware* from SoftwareLifeCycle (3.2.9).	False

Name	Type	Multiplicity	Description	Is derived
paramaterDataItemFile	Boolean	[1]	See *parameterDataItemFile* from SoftwareLifeCycle (3.2.9).	False
deactivatedCode	Boolean	[1]	See *deactivatedCode* from SoftwareLifeCycle (3.2.9).	False

Associations

Name	Type	Multiplicity	Description	Opposed member end
isComposedOf	Association	[1..*]	See *isComposedOf* from SoftwareLifeCycle (3.2.9).	«Process»

Constraints

See *SoftwareLifeCycle* (3.2.9).

4.3.10 *«SoftwareLifeCycleData»*

Description

The «SoftwareLifeCycleData» stereotype represents the documents that compile the data that have to be produced during the processes of the software life cycle. Such data are used to obtain certification of the software product and for post certification changes occurring to the software. Examples of produced data include requirements specification (SRD), requirements standards (SRS), plan for software aspects of certification (PSAC), software development plan (SDP), bug reports, test results (SVR), etc. DO-178C does not impose a specific form for the representation of these data.

Related concept

See *SoftwareLifeCycleData* (3.2.10).

Extensions

Base Metaclass	**Explanation**
Artifact	Artifact is the specification of a physical piece of information that is used or produced by a software development process. As such, «SoftwareLifeCycleData» represents any of the development deliverables that are required by DO-178C.

Attributes

Name	**Type**	**Multiplicity**	**Description**	**Is derived**
type	SoftwareLifeCycleDataType	[1]	See *type* from SoftwareLifeCycleData (3.2.10).	False

Name	Type	Multiplicity	Description	Is derived
controlCategory	ControlCategory	[1]	See *controlCategory* from SoftwareLifeCycleData (3.2.10).	False

Associations

Name	Type	Multiplicity	Description	Opposed member end
isProducedBy	Association	[1]	See *isProcudedBy* from SoftwareLifeCycleData (3.2.10).	«Activity»

4.3.11 *«SoftwareTestEnvironment»*

Description

The «SoftwareTestEnvironment» stereotype specifies various information related to the used testing environments.

Related concept

See *SoftwareTestEnvironment* (3.2.11).

Generalizations

Parent class name	Explanation
«Environment»	A software test environment is a specialization of the «Environment» stereotype.

Attributes

Name	Type	Multiplicity	Description	Is derived
differenceWithTargetComputer	String	[1..*]	See *differenceWithTargetComputer* from SoftwareTestEnvironment (3.2.11).	False
testTargetPlatform	testTarget	[1]	See *testTargetPlatform* from SoftwareTestEnvironment (3.2.11).	False
testTargetDescription	String	[1]	See *testTargetDescription* from SoftwareTestEnvironment (3.2.11).	False

4.3.12 *«TransitionCriterion»*

Description

The «TransitionCriterion» stereotype represents the minimum conditions defined by the software planning process to be satisfied in order to enter a process (RTCA,2011a).

Related concept

See *TransitionCriterion* (3.2.12).

Extensions

Base Metaclass	Explanation
Class	The metaclass Class allows us to represent any object without capturing unnecessary semantic of the UML metamodel that is not relevant for the introduced stereotype.

Attributes

Name	Type	Multiplicity	Description	Is derived
description	String	[1]	See *description* from TransitionCriterion (3.2.12).	False
isSatisfied	Boolean	[1]	See *isSatisfied* from TransitionCriterion (3.2.12).	False
condition	String	[1..*]	See *condition* from TransitionCriterion (3.2.12).	False
transitFromIncompleteProcess	Boolean	[1]	See *transitFromIncompleteProcess* from TransitionCriterion (3.2.12).	False

Name	Type	Multiplicity	Description	Is derived
allowReEntrance	Boolean	[1]	See *allowReEntrance* from TransitionCriterion (3.2.12).	False

Associations

Name	Type	Multiplicity	Description	Opposed member end
transitionTo	Association	[1..*]	See *transitionTo* from TransitionCriterion (3.2.12).	«Process»

4.4 Requirements Package

The elements of the conceptual model that are related to the software requirements process depicted in Figure 3.3 have been mapped to the UML metamodel to produce the Requirements package of the profile. Figure 4.4 depicts this package.

We remind the reader that we included the concept low-level requirement that pertains to the specification of the software requirements.

The stereotypes of the Requirements package are the following:

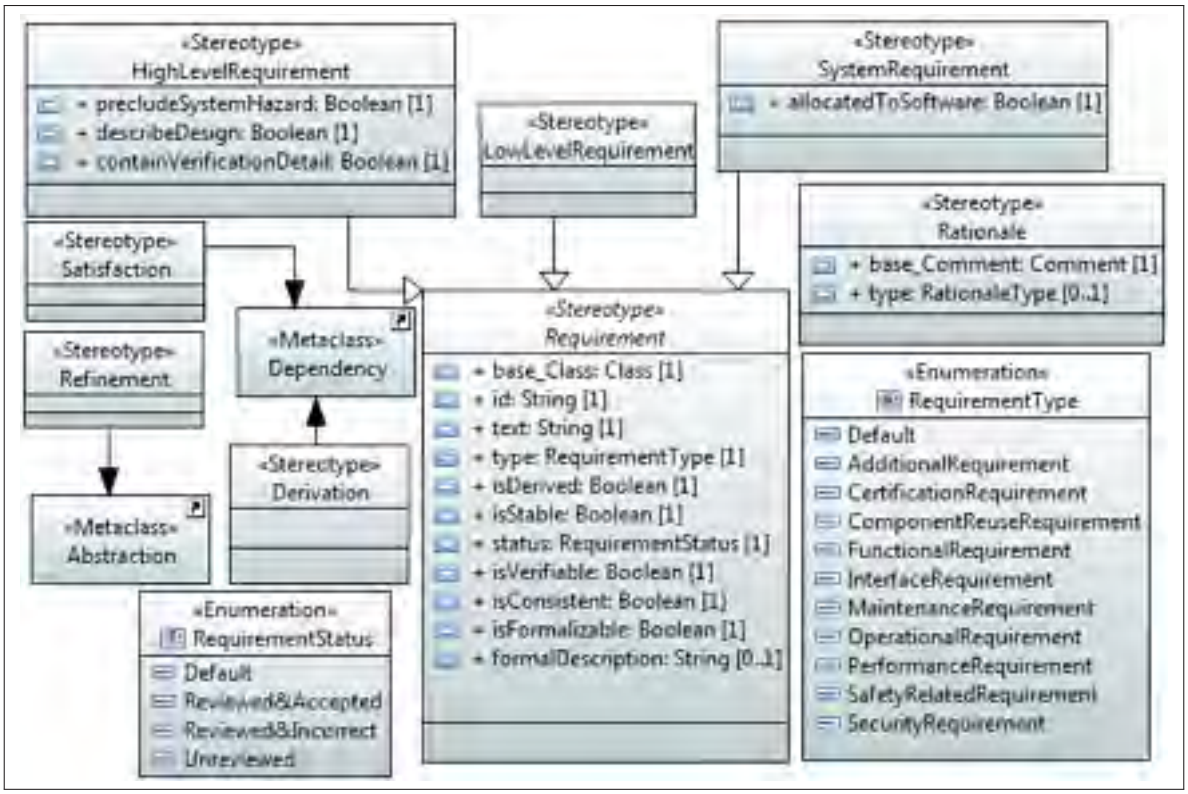


Figure 4.4 Requirements package diagram.

4.4.1 *«Derivation»*

Description

The «Derivation» stereotype is a relationship that enables the traceability of derived requirements. It related the derived requirement and the requirement being derived through a bi-directional trace.

The «Derivation» stereotype represents the relationship derivation from the conceptual model.

Related concept

See *Requirement* (3.3.4).

Extensions

Base Metaclass	Explanation
Dependency	A derivation relationship is used to indicate that a requirement add missing specification details from another. The metaclass Dependency enables the traceability that the derivation relationship introduces between two requirements.

Constraints

Constraint description	Software level	Introduced by supplement
A derivation relationship relates two requirements of the same level.	Software level D	None
A derivation relationship relates two requirements, one of them must have its attribute *isDerived* set to "True" while the second must have its attribute *isDerived* set to "False".	Software level D	None

4.4.2 *«HighLevelRequirement»*

Description

The «HighLevelRequirement» (HLR) stereotype captures software requirements that are developed from the analysis of system requirements that allocated to software, safety-related requirements, and system architecture (RTCA, 2011a).

Related concept

See *HighLevelRequirement* (3.3.1).

Generalizations

Parent class name	Explanation
«Requirement»	A High-level requirement is a specialization of the «Requirement» stereotype.

Attributes

Name	Type	Multiplicity	Description	Is derived
precludeSystemHazard	Boolean	[1]	See *precludeSystemHazard* from HighLevelRequirement (3.3.1).	False
describeDesign	Boolean	[1]	See *describeDesign* from HighLevelRequirement (3.3.1).	False
containVerificationDetail	Boolean	[1]	See *containVerificationDetail* from HighLevelRequirement (3.3.1).	False

Constraints

See *HighLevelRequirement* (3.3.1).

4.4.3 *«LowLevelRequirement»*

Description

The "LowLevelRequirement" (LLR) stereotype captures software requirements that are developed from the high-level requirements, derived requirements, and design constraints. Low-level requirements are requirements from which source code can be directly implemented without further information (RTCA, 2011a).

Related concept

See *LowLevelRequirement* (3.3.2).

Generalizations

Parent class name	Explanation
«Requirement»	A low-level requirement is a specialization of the «Requirement» stereotype.

Constraints

See *LowLevelRequirement* (3.3.2).

4.4.4 *«Rationale»*

Description

The «Rationale» stereotype purpose is to provide a justification for the decisions made during the software life cycle.

Related concept

See *Rationale* (3.3.3).

Extensions

Base Metaclass	Explanation
Comment	A «Rationale» could be attached to any of the model elements to justify some decision. The metaclass Comment can be attached to any metaclass of the UML metamodel.

Attributes

Name	Type	Multiplicity	Description	Is derived
type	RationaleType	[1]	See *type* from Rationale (3.3.3).	False

Constraints

See *Rationale* (3.3.3).

4.4.5 *«Refinement»*

Description

The «Refinement» stereotype is a relationship that enables the bi-directional traceability of a requirement decomposition into successive lower levels of requirement.

The «Refinement» stereotype has been created from the relationship refinement in the conceptual model.

Related concept

See *Requirement* (3.3.4).

Extensions

Base Metaclass	Explanation
Absraction	A refinement indicates a decomposition of a requirement into a lower level of specification. The metaclass Abstraction enables the traceability that the refinement relationship introduces between two requirements.

Constraints

Constraint description	Software level	Introduced by supplement
A refinement relationship relates two requirements.	Software level D	None
The following are the allowed relations that a refinement relationship may define between two requirements: • «SystemRequirement» ⟷ «HighLevelRequirement» • «HighLevelRequirement» ⟷ «LowLevelRequirement» • «LowLevelRequirement» ⟷ «LowLevelRequirement»	Software level D	None

4.4.6 *«Requirement»*

Description

The «Requirement» stereotype is an abstract stereotype that generalizes all kind of requirements defined by DO-178C (i.e. SRATS,HLR and LLR). A requirement's purpose is to describe what is to be performed by the system, or the software given a set of inputs and constraints (RTCA, 2011a).

Related concept

See *Requirement* (3.3.4).

Extensions

Base Metaclass	Explanation
Class	The metaclass Class allows us to represent any object without capturing unnecessary semantic of the UML metamodel that is not relevant for the introduced stereotype.

Attributes

Name	Type	Multiplicity	Description	Is derived
id	String	[1]	See *id* from Requirement (3.3.4).	False
text	String	[1]	See *text* from Requirement (3.3.4).	False
type	RequirementType	[1]	See *type* from Requirement (3.3.4).	False

Name	Type	Multiplicity	Description	Is derived
isDerived	Boolean	[1]	See *isDerived* from Requirement (3.3.4).	false
isStable	Boolean	[1]	See *isStable* from Requirement (3.3.4).	False
status	RequirementStatus	[1]	See *status* from Requirement (3.3.4).	False
isVerifiable	Boolean	[1]	See *isVerifiable* from Requirement (3.3.4).	False
isConsistent	Boolean	[1]	See *isConsistent* from Requirement (3.3.4).	False
isFormalizable	Boolean	[1]	See *isFormalizable* from Requirement (3.3.4).	False
formalDescription	String	[0..1]	See *formalDescription* from Requirement (3.3.4).	False

Constraints

See *Requirement* (3.3.4).

4.4.7 *«Satisfaction»*

Description

The «Satisfaction» stereotype is a relationship that enables the traceability between a requirement and the model elements that implement the said requirement.

The «Satisfaction» stereotype has been created from the relationship satisfaction in the conceptual model.

Related concept

See *Requirement* (3.3.4).

Extensions

Base Metaclass	**Explanation**
Dependency	A derivation relationship is used to indicate that a requirement add missing specification details from another. The metaclass Dependency enables the traceability that the derivation relationship introduces between two requirements.

Constraints

Constraint description	**Software level**	**Introduced by supplement**
A satisfaction relationship relates a high-level requirement or a low-level requirement to any element of the design.	Software level D	None

4.4.8 *«SystemRequirement»*

Description

The "SystemRequirement" (SRAT) stereotype captures requirements that describe at the system level the functionality that the system, as a whole, must fulfill in order to satisfy the stakeholders needs.

Related concept

See *SystemRequirement* (3.3.5).

Generalizations

Parent class name	Explanation
«Requirement»	A system requirement is a specialization of the «Requirement» stereotype.

Attributes

Name	Type	Multiplicity	Description	Is derived
allocatedToSoftware	Boolean	[1]	See *allocatedToSoftware* from SystemRequirement (3.3.5).	False

4.5 Verification Package

The elements of the conceptual model that are related to to the software verification process depicted in Figure 3.4 have been mapped to the UML metamodel to produce the Verification package of the profile. Figure 4.5 depicts this package.

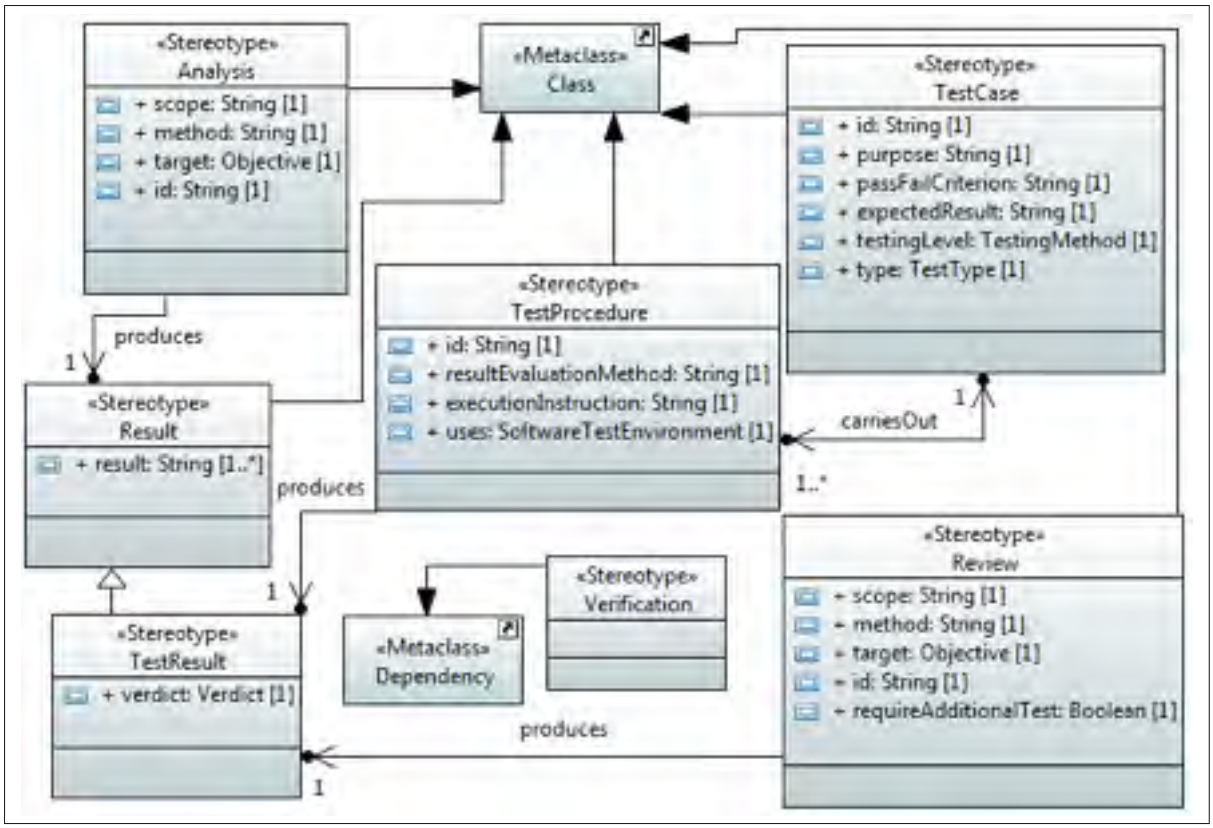


Figure 4.5 Verification profile diagram.

The stereotypes of the Verification Package are the following:

4.5.1 *«Analysis»*

Description

The «Analysis» stereotype is used to define an analysis. They provide repeatable evidences of correctness for the element(s) under scrutiny.

Related concept

See *Analysis* (3.4.1).

Extensions

Base Metaclasse	Explanation
Class	The metaclass Class allows us to represent any object without capturing unnecessary semantic of the UML metamodel that is not relevant for the introduced stereotype.

Attributes

Name	Type	Multiplicity	Description	Is derived
id	String	[1]	See *id* from Analysis (3.4.1).	False
scope	String	[1]	See *scope* from Analysis (3.4.1).	False
method	String	[1]	See *method* from Analysis (3.4.1).	False
target	«Objective»	[1]	See *target* from Analysis (3.4.1).	False

Associations

Name	Type	Multiplicity	Description	Opposed member end
produces	Association	[1]	Specifies the result of the analysis.	«Result»

4.5.2 *«Result»*

Description

The «Result» stereotype captures the information resulting from an analysis or a review.

Related concept

See *Result* (3.4.2).

Extensions

Base Metaclasse	Explanation
Class	The class metaclass allows us to represent any object without capturing unnecessary semantic of the UML metamodel that is not relevant for the introduced stereotype.

Attributes

Name	Type	Multiplicity	Description	Is derived
result	String	[1..*]	See *result* from Result (3.4.2).	False

4.5.3 *«Review»*

Description

The «Review» stereotype defines a review. They are used to provide a qualitative assessment of correctness for the element(s) under scrutiny. Requirements, test cases, test procedures, analyses and reviews can be reviewed.

Related concept

See *Review* (3.4.3).

Extensions

Base Metaclasse	Explanation
Class	The metaclass Class allows us to represent any object without capturing unnecessary semantic of the UML metamodel that is not relevant for the introduced stereotype.

Attributes

Name	Type	Multiplicity	Description	Is derived
id	String	[1]	See *id* from Review (3.4.3).	False
scope	String	[1]	See *scope* from Review (3.4.3).	False
method	String	[1]	See *method* from Review (3.4.3).	False

Name	Type	Multiplicity	Description	Is derived
target	«Objective»	[1]	See *target* from Review (3.4.3).	False
requireAdditionalTest	Boolean	[1]	See *requireAdditionalTest* from Review (3.4.3).	False

Associations

Name	Type	Multiplicity	Description	Opposed member end
produces	Association	[1]	Specifies the result of the review.	«Result»

4.5.4 *«TestCase»*

Description

The «TestCase» is a stereotype that represents the set of inputs, execution conditions and expected results developed for a particular testing objective. Examples of testing objectives include the execution of a specific program path or the verification of compliance against a specific requirement.

Related concept

See *TestCase* (3.4.4).

Extensions

Base Metaclasse	Explanation
Class	The class metaclass allows us to represent any object without capturing unnecessary semantic of the UML metamodel that is not relevant for the introduced stereotype.

Attributes

Name	Type	Multiplicity	Description	Is derived
id	String	[1]	See *id* from TestCase (3.4.4).	False
purpose	String	[1]	See *purpose* from TestCase (3.4.4).	False
passFailCriterion	String	[1]	See *passFailCriterion* from TestCase (3.4.4).	False
expectedResult	String	[1]	See *expectedResult* from TestCase (3.4.4).	False
testingLevel	TestingMethod	[1]	See *testingLevel* from TestCase (3.4.4).	False
type	TestType	[1]	See *type* from TestCase (3.4.4).	False

Associations

Name	Type	Multiplicity	Description	Opposed member end
carriesOut	Association	[1..*]	See *carriesOut* from TestCase (3.4.4).	«TestProcedure»

4.5.5 *«TestProcedure»*

Description

The «TestProcedure» stereotype defines a test procedure. A test procedure provides the detailed instructions for the set-up and execution of a given test case, along with the instructions required for the evaluation of the related test case execution results.

Related concept

See *TestProcedure* (3.4.5).

Extensions

Base Metaclasse	Explanation
Class	The metaclass Class allows us to represent any object without capturing unnecessary semantic of the UML metamodel that is not relevant for the introduced stereotype.

Attributes

Name	Type	Multiplicity	Description	Is derived
id	String	[1]	See *id* from TestProcedure (3.4.5).	False

Name	Type	Multiplicity	Description	Is derived
executionInstruction	String	[1]	See *executionInstruction* from TestProcedure (3.4.5).	False
resultEvaluationMethod	String	[1]	See *resultEvaluationMethod* from TestProcedure (3.4.5).	False
uses	«SoftwareTestEnvironment»	[1]	See *uses* from TestProcedure (3.4.5).	False

Associations

Name	Type	Multiplicity	Description	Opposed member end
carriesOut	Association	[1]	See *carriesOut* from TestProcedure (3.4.5).	«TestCase»
produces	Association	[1]	See *procudes* from TestProcedure (3.4.5).	«TestResult»

4.5.6 *«TestResult»*

Description

The «TestResult» stereotype provides the resulting information related to a test case execution. The test results are meant to be compared with the expected test results provided by the test case in order to evaluate the results of the execution of the related test case.

Related concept

See *TestResult* 3.4.6.

Genezalizations

Parent Class name	Explanation
«Result»	A test result is a specialization of the «Result» stereotype.

Attributes

Name	Type	Multiplicity	Description	Is derived
verdict	Verdict	[1]	See *verdict* from TestResult (3.4.6).	False

4.5.7 *«Verification»*

Description

The «Verification» stereotype defines the relationship that is used as a traceability mean between test cases, test procedures, review or analysis and the element under verification.

Related concept

See *TestCase*, *Analysis* and *Review*.

Extensions

Base Metaclasse	**Explanation**
Dependency	A derivation relationship is used to indicate that a requirement add missing specification details from another. The metaclass Dependency enables the traceability that the derivation relationship introduces between two requirements.

Constraints

Constraint description	**Software level**	**Introduced by supplement**
The client of a verification relationship should be a test case, a review or an analysis.	Software level D	None

CHAPTER 5

CASE STUDY - THE LANDING GEAR CONTROL SOFTWARE

To demonstrate the usefulness of the proposed profile, we applied the profile to a realistic case study: the landing gear control software (LGCS). The LGCS (Paz & El Boussaidi, 2017) has been developed and adapted from the case study proposed by Boniol & Wiels (2014). The objective of this case study is to develop a specification that is consistent with the practices related to safety-critical software development in the avionic domain. It specifically aims at being compliant with DO-178C. For this purpose, this case study has been developed by following the best practices compiled in the requirements engineering handbook (Lempia & Miller, 2009) and in respect of the guidelines provided by DO-178C and its supplements.

This chapter is organized as follows. Section 5.1 presents the implementation of the profile within an UML modeling environment. In Section 5.2, we demonstrate how the profile can be used to support the modeling of the LGCS case study. In particular, we illustrate the usefulness of our profile through the usage scenarios that were identified in Chapter 2 (Section 2.3).

5.1 Tool support

To provide an appropriate support for implementing our profile, we identified the following as the requirements that have to be satisfied by a UML modeling tool:

- Allow the creation of UML profiles and their integration within the tool.
- Allow the creation of new diagram types to support the specification diagram, the planning diagram and the verification diagram introduced by our profile.
- Support the validation of constraints to verify the correctness the profile's stereotypes usage.
- Allow the customization of the messages resulting from violated constraints.

In the following, we describe briefly the UML tool we used to implement our profile and the steps of the implementation process.

5.1.1 Papyrus

Among the existing tools that meet the aforementioned requirements, we have chosen Papyrus [1] as the UML environment in which we implement our profile. Papyrus is an open source, general purpose UML modeling tool. Papyrus is compliant with the UML 2.5 specification, as such it provides support for all UML diagrams including the profile diagram. Papyrus is built upon the Eclipse Modeling Framework (EMF). Papyrus is provided as a set of plug-ins to leverage Eclipse software components in order to provide an efficient environment for modeling activities using the UML language. It provides support for source code generation (Java, C++) from UML models. It also provides model validation through EMF validation framework. Furthermore, it provides support for other OMG standards such as SysML, MARTE, BPMN, fUML, and PSCS. Finally, it facilitates the implementation of DSMLs based on UML profiles.

In order to ease the integration of DSMLs, Papyrus provides a set of extension points that can be used by profile designers to integrate their profiles within Papyrus. Eclipse plug-ins are the encapsulation of a set of behaviors that interact with each other in order to form a new running environment. Plug-ins are built in order to extend the behavior of the eclipse platform using various extension points to register their customized behavior. Extension points are the interfaces provided by a plug-in to allow new plug-ins to further customize or extend the behavior of the environment. An extension point is a contract that the extending plug-in conform to.

In the context of our work, we have been working with Papyrus "Neon" release (version 2.0.x).

To implement and integrate our profile within the Papyrus environment we have studied the implementation of the SysML profile provided by Papyrus[2]. As an additional resource, the

[1] Papyrus: https://eclipse.org/papyrus/

[2] The SysML Papyrus git: http://git.eclipse.org/c/papyrus/org.eclipse.papyrus-sysml.git/

developers of Papyrus provide a library project example [3] that serves as an example for the definition and the integration of a UML profile within papyrus.

5.1.2 Implementing the DO-178C profile

Papyrus offers a built-in mechanism to create and integrate a UML profile within its environment. The process of implementing a UML profile within Papyrus can be decomposed into two steps. The first consists in creating the profile model. The second step consists in creating the plug-ins that are in charge of the integration of the profile within the modeling environment.

5.1.2.1 Step 1: Creation of the Profile Model

To create a UML profile, Papyrus provides a profile diagram. The profile diagram allows designers to create stereotypes, declare their attributes, the relationships that exist between stereotypes and to specify the base metaclasses of stereotypes. Figure 5.1 shows the Papyrus wizard for creating new profile. Figure 5.2 shows the Papyrus environment workbench with the profile diagram view opened showing the Requirements package of the DO-178C profile.

Figure 5.1 Papyrus create new model wizard.

[3] The library profile example: http://git.eclipse.org/c/papyrus/org.eclipse.papyrus.git/tree/examples/library?h=streams/2.0-maintenance

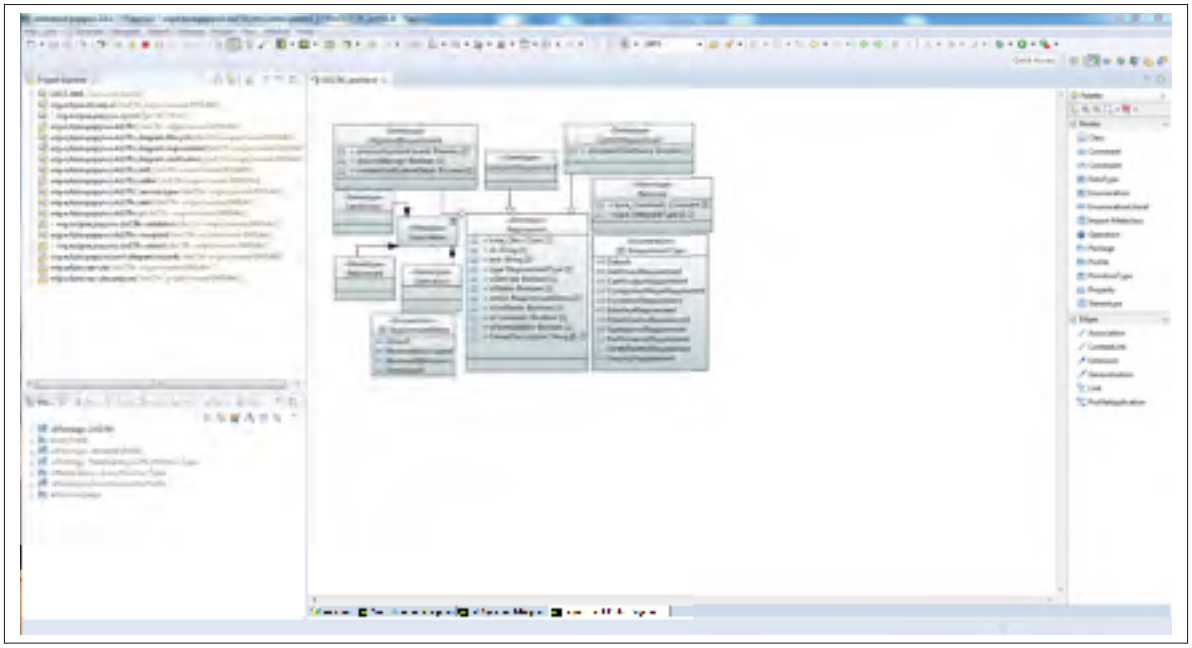


Figure 5.2 Papyrus interface, profile diagram view.

5.1.2.2 Step 2: Integrating the profile within the Papyrus tool

To make the profile available as part of the Papyrus tool, we have to create a number of Eclipse plug-ins which extend and customize Papyrus plug-ins. We organized these plug-ins into a structure that is similar to the one used in the SysML profile project. The created plug-ins are organized according to the following structure:

- **Core**: The core folder contains the plug-ins that are used to register a profile within Papyrus. Their purpose is to provide the environment with the definition of the profile, the profile's constraints and the necessary constructs to manipulate the concepts of the proposed profile as EMF model.
- **Diagram**: The diagram folder contains the plug-ins that are in charge of the definition of new diagrams, their associated graphical notations and restrictions within the Papyrus environment.

- **GUI**: The GUI folder contains the plug-ins that have an impact on the environment in term of the graphical interface such as the property view for a profile's stereotype and icons used in various menus.
- **Papyrus**: The Papyrus folder contains the plug-ins that are modified versions of the plug-ins as distributed by Papyrus.

Thus, once the profile diagram is created, we use Papyrus to generate the implementation of the entities contained in the profile. Papyrus relies on the EMF code generator. The generated code is part of the `Core` folder. To use the generated implementation of the profile, we identified the profile as a namespace and registered that namespace in the EMF global package registry. This is done through extension points defined by Papyrus.

Once we registered the profile, we defined extensions to customize the graphical interface of Papyrus to enable the user to select the profile, visualize the profile's specific viewpoint and create the profile's diagrams. It is worth mentioning that we defined four templates within our profile to customize the software life cycle diagram according to the software level. In other words, the profile user can choose a given template and the profile will create an initialized software life cycle diagram which conforms to the software level of the chosen template.

Once all extensions are done, the user can launch Papyrus and create a model that uses the DO-178C profile (Figure 5.3). The user can then select a template corresponding to the software level of his model or select directly the diagrams he wishes to include in his model (Figure 5.4). Figure 5.5 shows the DO-178C view with a requirement diagram. The details of this second step are described in Annex III.

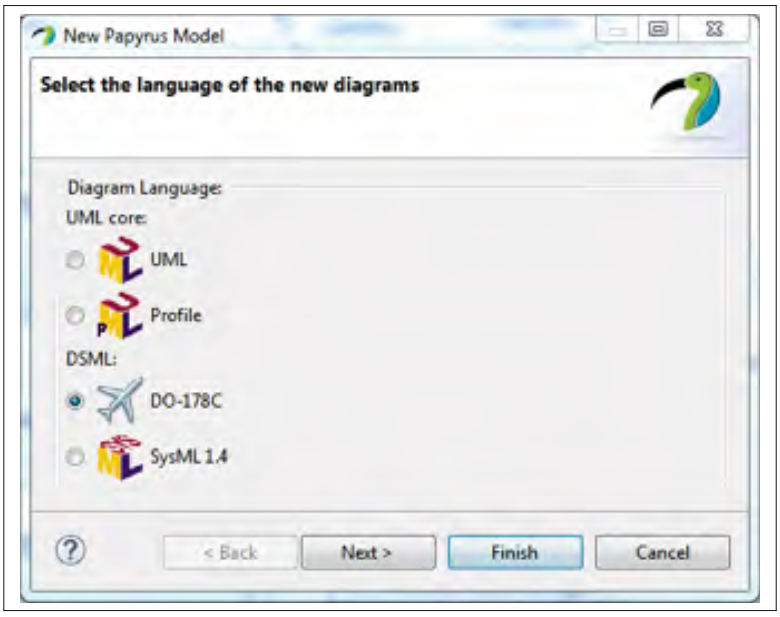


Figure 5.3 Papyrus create new model wizard selecting the DO-178C profile.

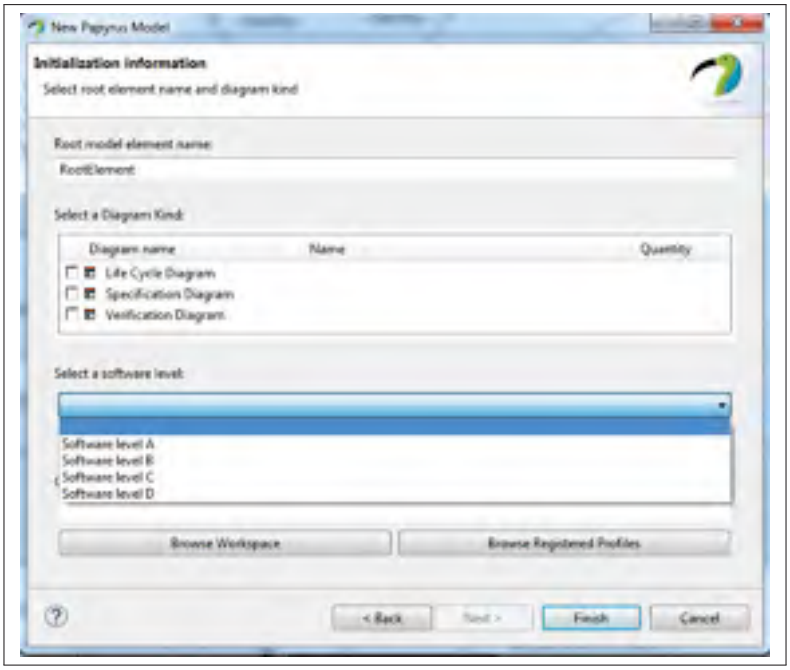


Figure 5.4 Papyrus create new model wizard selecting the diagram kind.

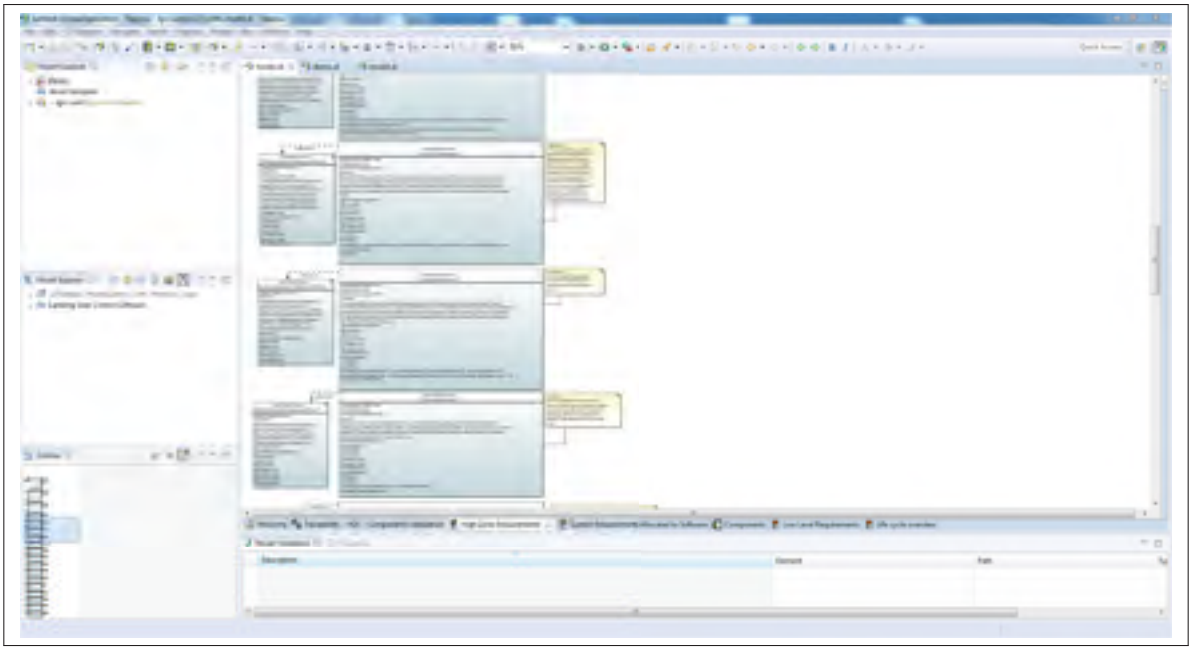


Figure 5.5 Papyrus view of the DO-178C requirement diagram.

5.2 Using the DO-178C profile to model an avionic software

In this section, we first give an overview of the avionic software that we used to assess the usefulness of the DO-178C profile. We illustrate the usage of the profile through four use case scenarios that were identified in Chapter 2.

5.2.1 Landing Gear Control Software - Overview

The landing gear control software (LGCS) is in charge of the actuation of an aircraft's landing gear system. A landing gear system is composed of three gears that are retractable in order to enable taking off and landing maneuvers of an aircraft. These wheels assemblies are arranged in a triangle configuration in order to support the aircraft's weight while it remains on the ground. Two of these assemblies are located under the wings and the remaining one is located under the aircraft's nose. Figure 5.6 shows the undercarriage configuration of a landing gear system as viewed from the front of the aircraft. Once the plane has taken off, each gear is retracted into its

respective compartment. These compartments, in which the gears are concealed, have doors that are opened and closed upon extension or retraction of the gears.

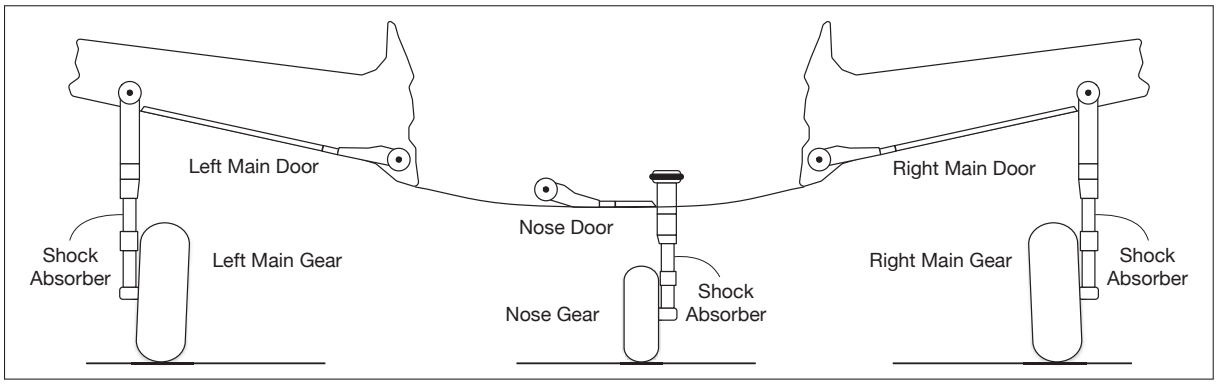


Figure 5.6 Front view of an aircraft undercarriage configuration. Extracted from Paz & El Boussaidi (2017).

There are two distinct sequences for the gears motion, the extension of the gears for the landing phase of the flight and its opposite sequence, retraction of the gears after take off. The extension of the gears allowing landing of the aircraft, consists in the opening of the gear's compartment doors followed by the extension of the gear. The opposite sequence happens once the aircraft has taken off and enters its climbing phase, the gears are retracted into their respective compartment followed by the closing of the compartment's doors. Figure 5.7, provides a view of the retraction sequence for one of the gears.

Actuation of the gears is performed through a hydraulic circuit that is controlled by a set of electro-valves that regulate the pressure in the segment of the circuit they control. The hydraulic circuit is controlled by five-electro-valves. One general electro-valve is in charge of the pressurization of the whole circuit while four specific electro-valves are in charge of further tuning the pressure in subsections of the circuit in order to open the doors, close the doors, extend the gears and retract the gears.

A pilot interface is provided in the cockpit to enter the desired motion of the gears. This interface is comprised of a lever that has two positions: up and down, respectively to retract the gears and

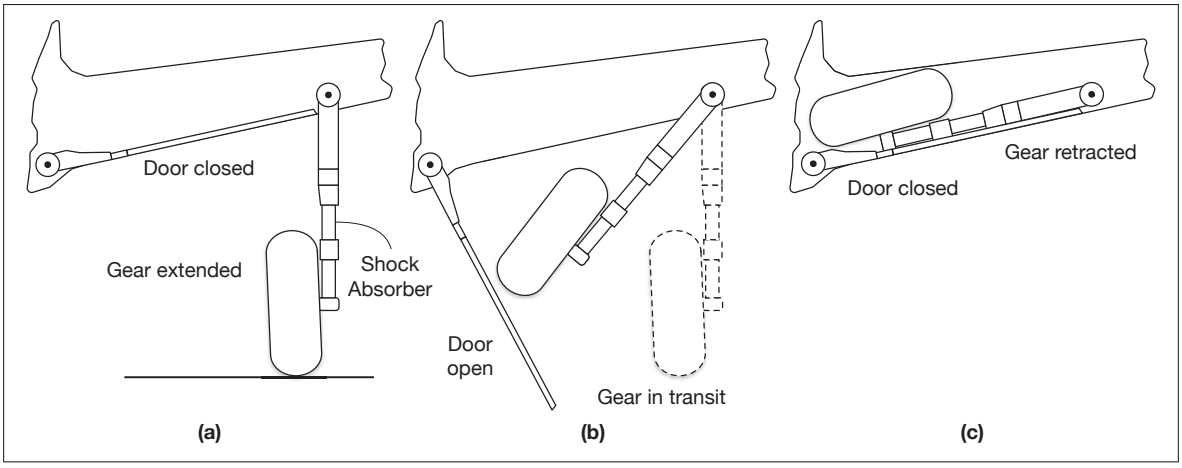


Figure 5.7 Phases of the retraction sequence: (a) extended gear, (b) gear in transit, and (c) retracted gear. Extracted from Paz & El Boussaidi (2017).

to extend the gears. The pilot indicates the desired motion to be performed by moving the lever up or down. To notify the pilot about the state of the system, the interface comes with a set of colored lights. A green light indicates that the system is functioning normally as defined by its requirements, an amber light indicates that the gears are in transit and the system is functioning normally, finally a red light indicates that a failure of the system has been recorded and the system is no longer functioning. To prevent accidental actuation requests, the system contains an analogical switch that enables and disables the stimulation of the electro-valves. The switch mechanically closes each time the position of the lever changes, thus enabling the subsequent stimulation of the electro-valves.

The landing gear system contains a set of 17 sensors for monitoring the state of each component of the system. These sensors include one sensor to monitor the analogical switch state, one sensor to read the pressure of the hydraulic circuit, two sensors per gear to evaluate their position, two sensors per door to determine their position, and three sensors (one per gear) to determine whether the gear shock absorbers are relaxed. Each sensor reading is performed three times and these readings are subdued to a voting process prior to determine the state and the value of each sensor reading.

The behavior of the physical system is controlled, and monitored by the landing gear controller software (LGCS). The controller is in charge of communicating with the entities that constitute the landing gear system. It is responsible for the monitoring activity of the system's state in order to provide the pilot with feedback about the current state of the system (e.g. gears being in transition, system failure). Upon request from the pilot, the controller will send specific commands to the system's electro-valves to perform the requested actuation of the landing gears. Figure 5.8 provides a contextual view of the various relations that exist between the system entities and the software controller.

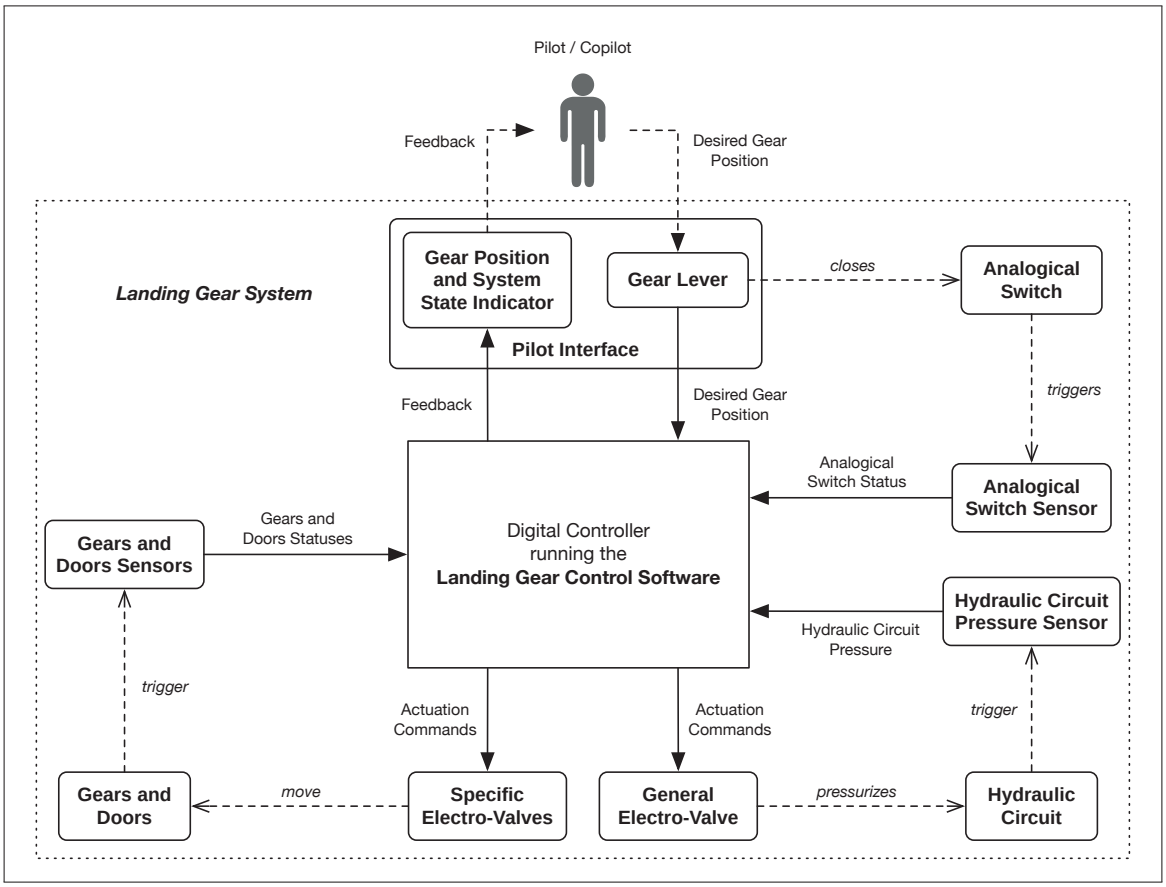


Figure 5.8 The LGCS operational context. Extracted from Paz & El Boussaidi (2017).

5.2.2 Use case 1: Specifying the software life cycle of the LGCS

The software planning process defines the means that are used in order to produce a software product that complies with its airworthiness requirements and the targeted software level. In order to specify the software life cycle that will drive the development of the LGCS, we use the stereotypes defined in the LifeCycle package of the proposed profile. These stereotypes and their associated constraints allow to specify the set of processes, objectives, and activities that together form a DO-178C compliant software life cycle for the LGCS targeted software level. Figure 5.9 shows the DO-178C profile viewpoint with an excerpt of the LGCS software life cycle. In the following we present in details the main concepts that we modeled in the LGCS software life cycle.

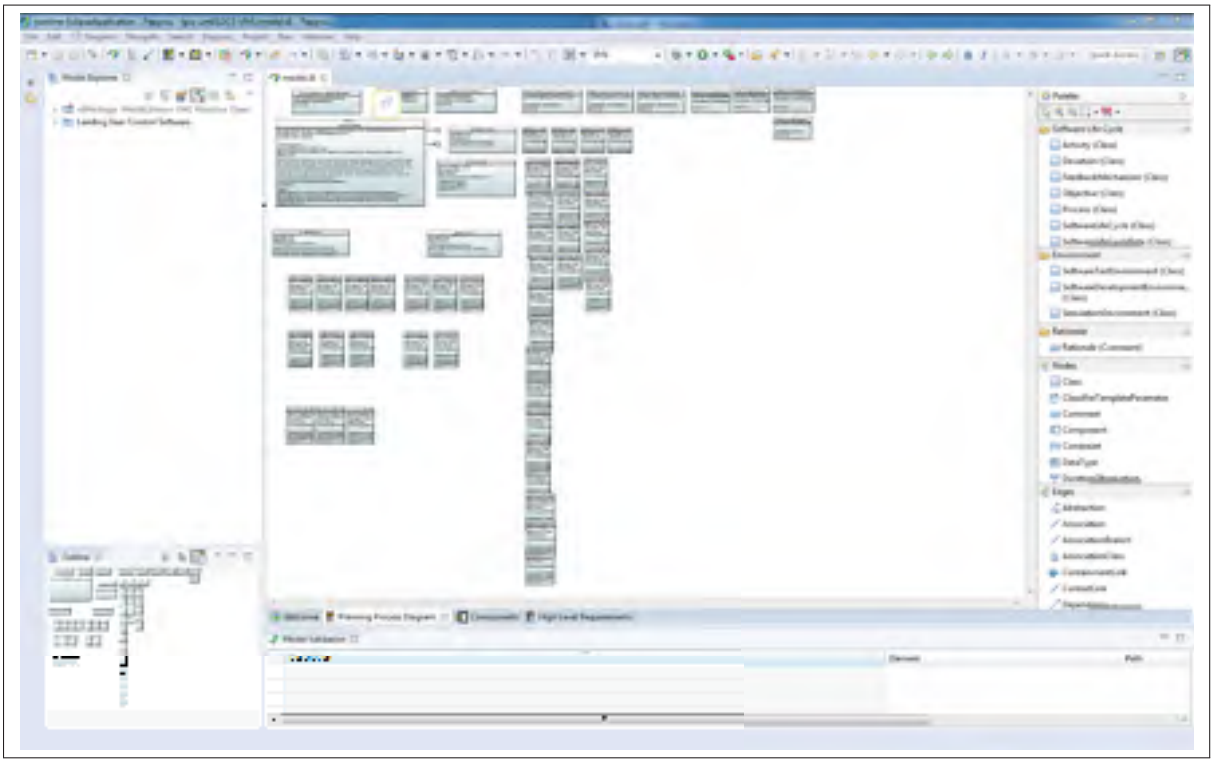


Figure 5.9 Example of a model of the software planning process.

The LGCS software life cycle is specified by using the «SoftwareLifeCyleStereotype» as observed in Figure 5.10. The development of the LGCS shall be performed by exercising the

activities that DO-178C defines for software that target an assurance level of C. Because the design of LGCS is done using UML, the development of the LGCs is guided by the guidelines provided by DO-331 and DO-332 respectively dealing with model-based development and object-oriented technologies. The specification of the target software level and the applicable supplements enable the profile to adjust the set of constraints that have to be verified in order to validate the model. The LGCS software life cycle is composed of the following processes: planning process, software requirements process, software design process, software coding process, integration process, software verification process, software quality assurance process, software configuration management process, and the certification liaison process. Additionally, the LGCS software life cycle captures further information that impact the evidence that have to be produced for certification of the software product. Among those, the software requirements for the LGCS are defined using multiple levels of requirements meaning that both HLRs and LLRs will be developed from the system requirements allocated to software. Additionally, the LGCS does not use parameter data item files. The LGCS is not a multiple version dissimilar software. For its development, the LGCS does not use previously developed software. Finally, the LGCS is not a user modifiable software. As such no additional certification evidence is required in relation to these properties of the software.

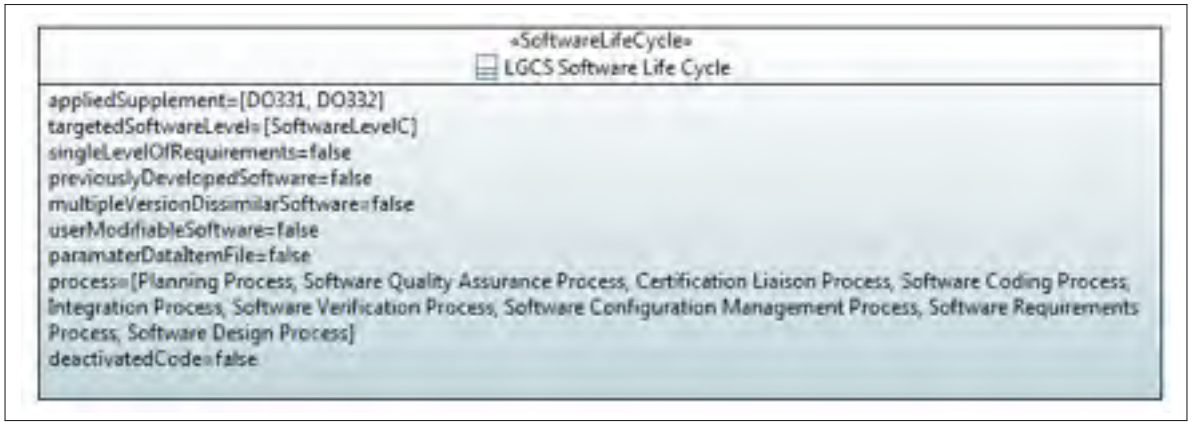


Figure 5.10 Specification of the software life cycle.

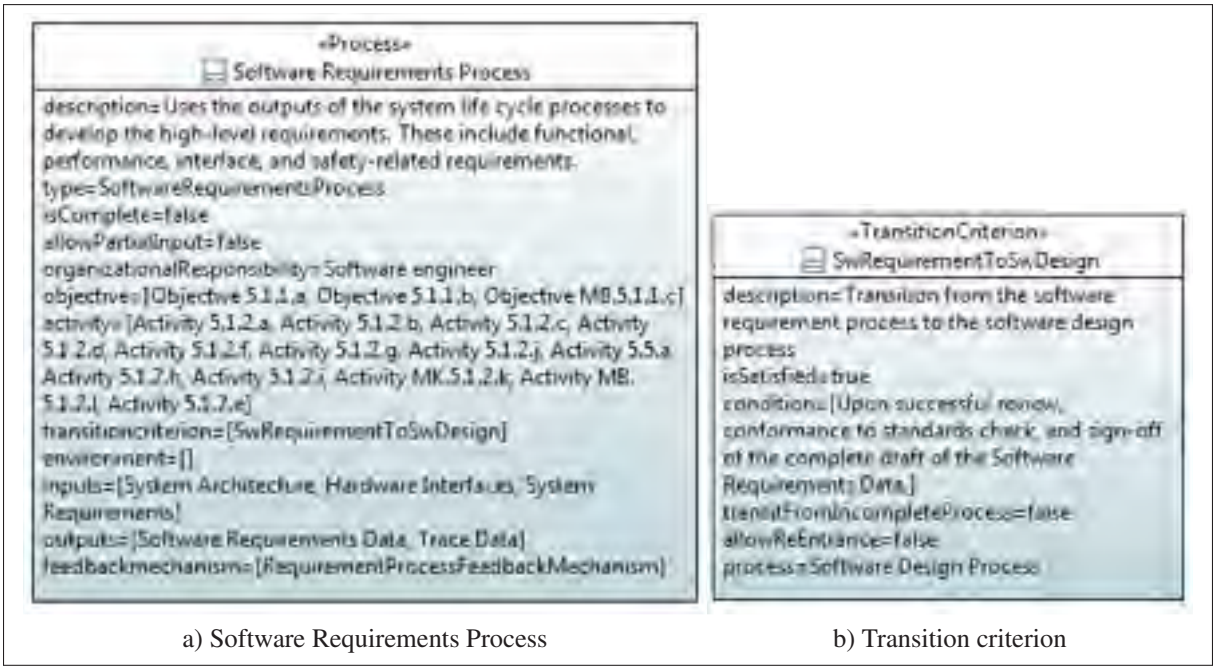
The software planning process as defined for the LGCS is provided in Figure 5.11. The planning process is responsible for the definition of the complete software life cycle guiding the development of the LGCS, thus defining the activities and objectives to be met during the development of the LGCS for each of the processes that constitute the software life cycle. The planning process does not receive data as input, however it produces outputs that are used for demonstrating compliance in order to certify the software product. The documents gathering the data produced during the planning are the software verification plan (SVP), the plan for software aspects of certification (PSAC), the software development plan (SDP), the software configuration management plan (SCMP), the software quality assurance plan (SQAP), the software design standards (SDS), the software code standards (SCS), the software requirements standards (SRS), the software verification results (SVS), and the software model standards (SMS). The software planning process is deemed complete upon approval of the PSAC, the SDS, the SDP, and the SVP by the certifying authorities. Upon completion of the software planning process, the specification of the software requirements shall begin as part of the software requirements process. The project leader is responsible for the activities of the planning process that result in the specification of the LGCS life cycle.

a) Software Planning Process

b) Transition criterion

Figure 5.11 The software planning process and its apportioned transition criterion.

Figure 5.12 provides the definition of the software requirement process as defined for the LGCS and the transition criteria that applies to the software requirement process. The software requirement process shall lead to the definition of the high-level requirements that are to be contained in the Software Requirements Data. These HLRs are developed from the inputs received by the process (i.e. the system architecture, the hardware interfaces, and the system requirements allocated to software). The requirement process has 3 objectives. Among those, objective 5.1.1.a (depicted in Figure 5.13.a) is to develop the high-level requirements and is applicable for any software level higher than software level D. These objectives are achieved by performing 13 activities, among which activity 5.1.2.c (depicted in Figure 5.13.b) states that each SRATS have to be specified in the HLRs. Upon completion of the activities of the software requirements process, transition to the software design process can be done only once reviews and conformance checks against the software requirements standards are performed.



a) Software Requirements Process b) Transition criterion

Figure 5.12 The software requirement process and its apportioned transition criterion.

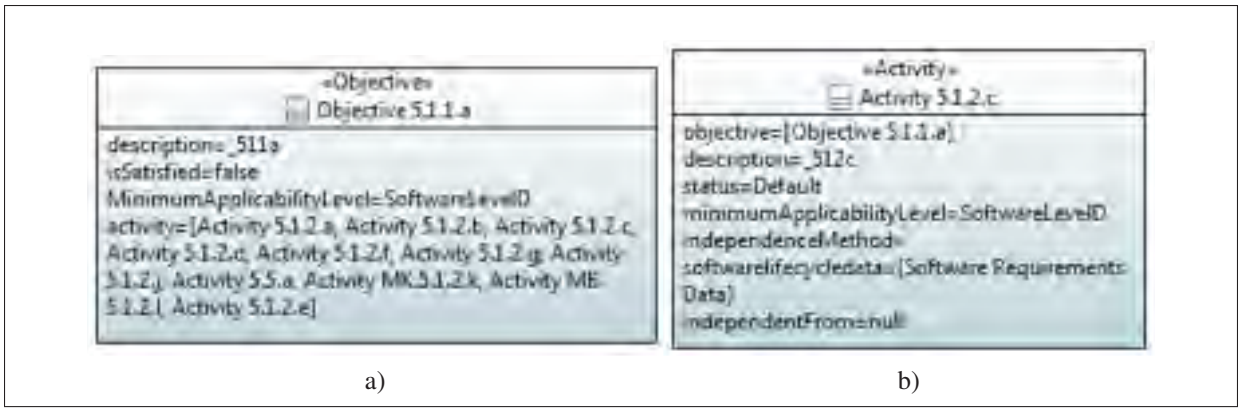


Figure 5.13 An objective (a) and an activity (b) of the software requirement process.

The profile defines constraints to help in the specification of the entities that constitute the software life cycle. Constraints verify the correct assignment of objectives to a process and the definition of the activities that are related to an objective. Furthermore the constraints also verify that the correct documents required by DO-178C are received and emitted by the relevant processes. Examples of these constraints are:

- Software level D: The software requirements process shall contain the activity "5.1.2.a". This constraint is applicable to software targeting software level D and higher.
- Software level D: The software requirements process shall contain the objective "5.1.1.a". This constraint is applicable to software targeting software level D and higher.
- Software level D: The software requirements process shall output the following document: Software Requirements Data. This constraint is applicable to software targeting software level D and higher.
- Software level D: The software requirements process shall receive the following input: System Requirements. This constraint is applicable to software targeting software level D and higher.

We modified the previous requirements process (Figure 5.12.a) to remove activity 5.1.2.a and objective 5.1.1.a from the process specification. We then launched the verification of the LGCS

model and the profile identified the violated constraints as shown in Figure 5.14. From the guidance provided by the violated constraint(s), the software designer may proceed to adjust the specified software life cycle in order to fill the missing information, or to perform the necessary correction in order to be compliant with DO-178C.

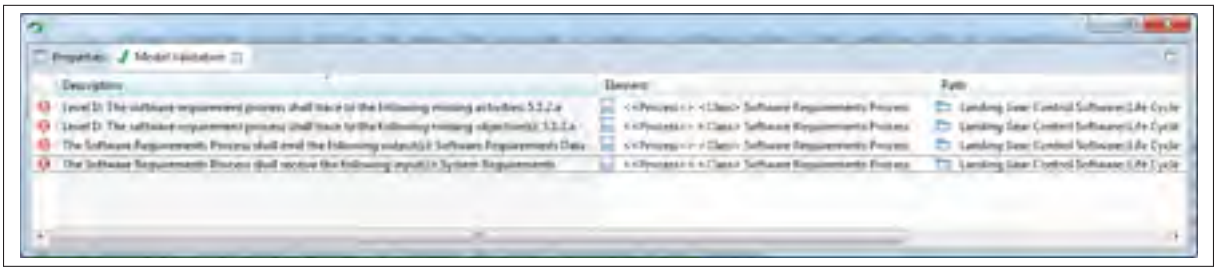
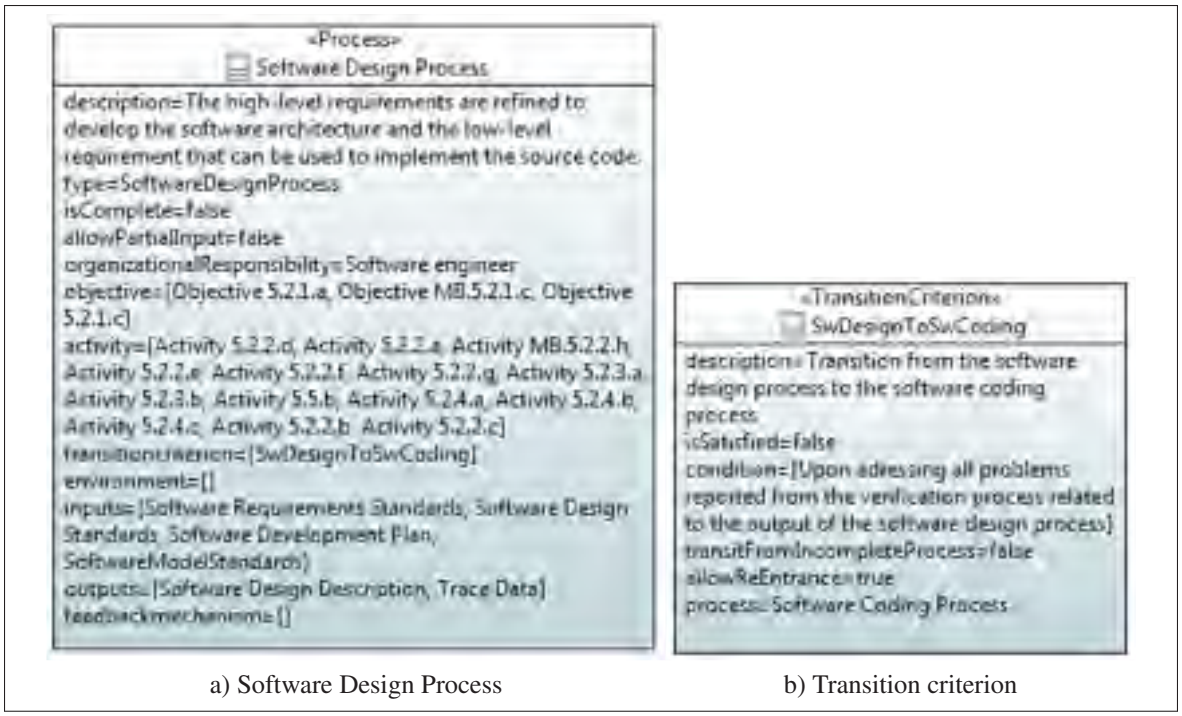


Figure 5.14 Examples of violated constraints for the software requirement process.

Figure 5.15 provides the specification of the LGCS software design process and its transition criteria. Upon completion of the design process, the software architecture and the low-level requirements are developed from the high-level requirements received as input in the software requirements data. The activities of the design process are placed under the responsibility of the software engineer. The design process is composed of 14 activities that achieve 3 objectives. The output of the design process includes the software architecture and the detailed description of the design. These are contained in the software design description document. The design process is deemed complete once all of the problems that relate to the outputs of the design process reported by the verification process are addressed.



a) Software Design Process b) Transition criterion

Figure 5.15 The software design process and its apportioned transition criterion.

Figure 5.16 provides the specification of the software verification process as defined for the LGCS. No transition criterion have been allocated to the software verification process because its activities are performed in parallel with the other processes of the software life cycle. It has 36 objectives. No activities are assigned to the process. This is due to the way DO-178C guidelines related to the software verification process are written. Indeed the document is written in a way that only declares the objectives of the process and not its activities. As such the objectives of the process are also its activities. The verification process receives data from the processes of the life cycle. Receiving the system requirements along with the software requirements (Software Requirements Data), the software architecture (Software Design Description), the source code, the executable object code and the Software verification Plan. The test cases and procedures that are developed and executed to verify the correct implementation of the software are defined in the Software Verification Cases and Procedures. The results of the executed test cases, the performed analyses and reviews are contained in the Software Verification Results.

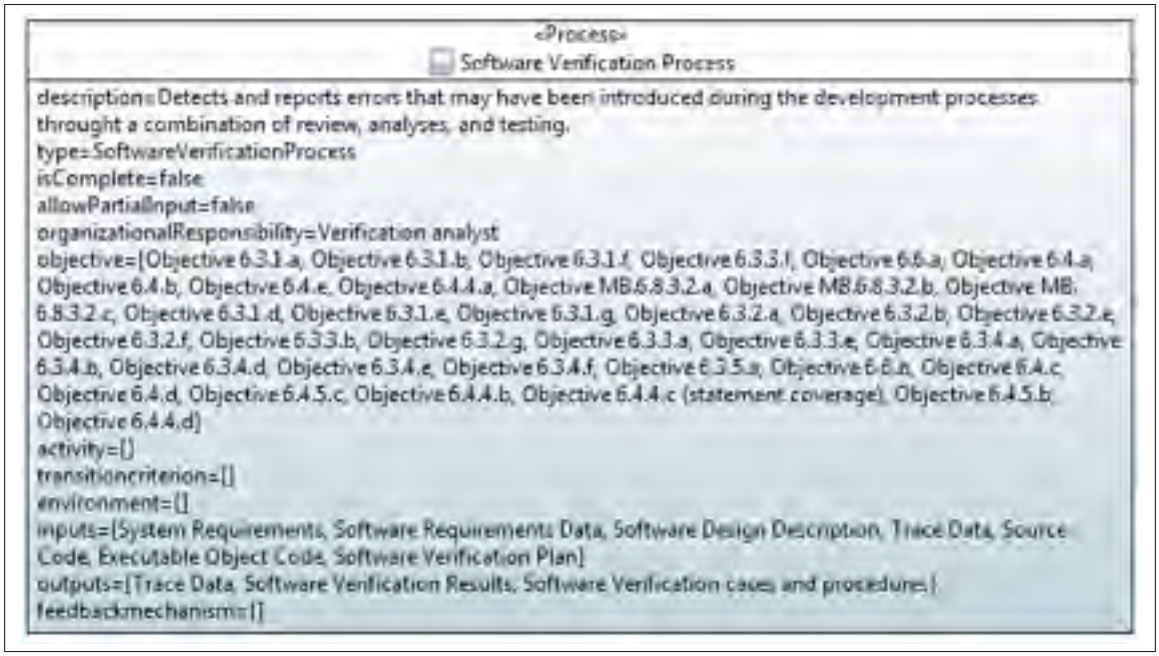


Figure 5.16 The software verification process.

5.2.3 Use case 2: Specifying requirements

Because the specification of the landing gear control software contains a large number of software requirements (18 HLRs and 76 LLRs), we focused on a subset of these requirements to show how the profile can be used to capture relevant DO-178C evidences. Thus, out of the 18 high-level requirements that are used to specify the landing gear controller, we used the five high-level requirements shown in table 5.1 and a subset of their refinement into low-level requirements (two) shown in Table 5.2.

Table 5.1 Subset of the LGCS High-level requirements.
Extracted from Paz & El Boussaidi (2017).

ID	Description	Traces
HLR-1	When the LGCS receives data from one of the LGCS sensors, the LGCS shall process the three `Readings` associated to the sensor data based on the following rules [...] (the rules are not shown due to the length of the constraint).	SRATS-1
HLR-4	When the LGCS is currently executing a retraction sequence and a `Down` value is received for the `Desired Gear Position`, the LGCS shall halt the current retraction sequence and revert all the actions that were executed. Likewise, when the LGCS is currently executing an extension sequence and an `Up` value is received for the `Desired Gear Position`, the LGCS shall halt the current extension sequence and revert all the actions that were executed.	SRATS-4 LLR-35
HLR-6	Once the overall value of the `Hydraulic Circuit Pressure` is greater than or equal to 30,000 kPa and less than 35,000 kPa after the `General EV Actuation Command` is set to `Open`, the lgcs can set to `Open` the necessary specific EV (i.e. `Door Closing EV Actuation Command`, `Door Opening EV Actuation Command`, `Gear Retraction EV Actuation Command` or `Gear Extension EV Actuation Command`).	SRATS-6 LLR-14 LLR-43 LLR-44 LLR-45
HLR-7	Once at least 0.2 seconds have elapsed since the `General EV Actuation Command` was set to `Open`, the LGCS can set to `Open` the `Door Opening EV Actuation Command`.	SRATS-7 LLR-14 LLR-46
HLR-12	Once 2 seconds have elapsed since the `General EV Actuation Command` was set to `Open` and the overall value of the `Hydraulic Circuit Pressure` is still less than 30,000 kPa, the LGCS shall detect a failure of the general hydraulic electro-valve and halt the currently executing sequence.	SRATS-12 LLR-44 LLR-56 LLR-57

Table 5.2 Subset of the LGCS low-level requirements.
Extracted from Paz & El Boussaidi (2017).

ID	Description	Refines
LLR-44	If the `waitForHydraulicPressure` method is active and the overall value of the `Hydraulic Circuit Pressure` monitorable variable is less than 30,000 kPa, the `waitForHydraulicPressure` method shall remain active until the `PressurizationTimeoutEvent` is raised.	HLR-6 HLR-12
LLR-56	If the `waitForHydraulicPressure` method is active and 2 seconds have elapsed since the `General EV Actuation Command` was set to `Open`, the `PressurizationTimeoutEvent` shall be raised.	HLR-12

Through its `Requirements` package the proposed profile allows to specify the system requirements that are allocated to software by using the «`SystemRequirement`» stereotype and to specify their refinement into high-level requirements using the «`HighLevelRequirement`» stereotype. Figure 5.17 shows the use of these two stereotypes along with the «`Refinement`» stereotype to specify the decomposition of a system requirement allocated to software into high level requirements. We can observe that in this example, the refined SRATS is simply refined into a high-level requirement without modification made to its text, meaning that the system requirement to be complete enough to be considered as a high-level requirement. HLR-4 does not provide details about the design and does not include verification details as prescribed by DO-178C.

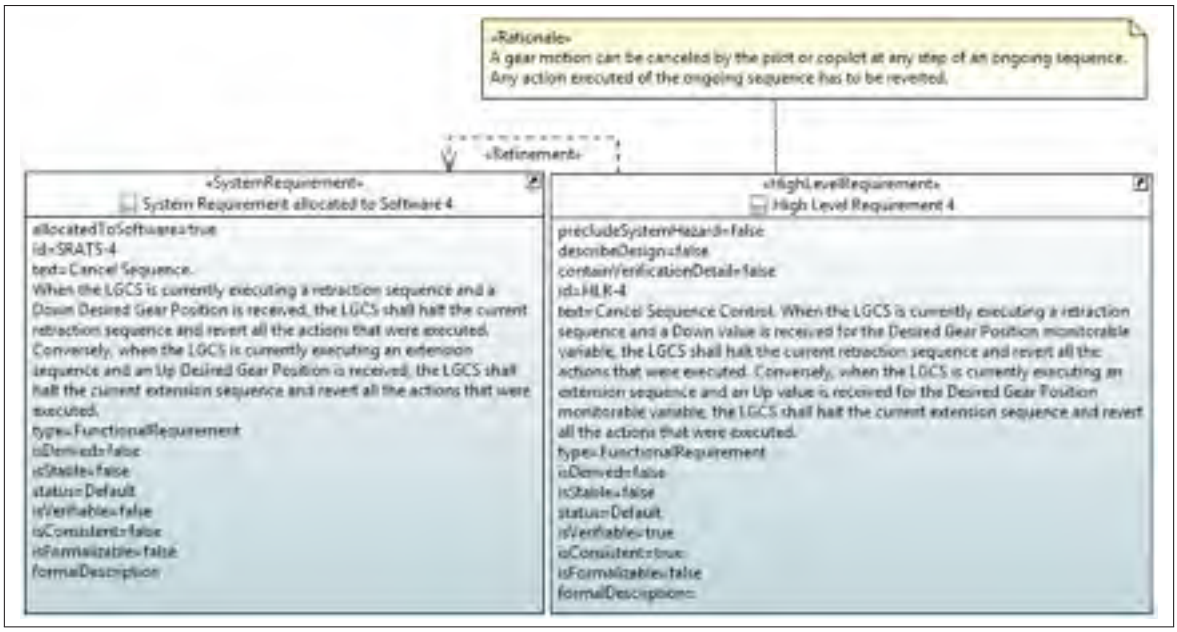


Figure 5.17 Refinement of a system requirement allocated to software into a high-level requirement.

In the context of the LGCS, we did not encounter any derived requirements. To show how the profile supports the specification of derived requirements, we use a generic example. Figure 5.18.a shows two HLRs that are related with the «derivation» association. The attributes `isDerived` is automatically computed by the environment based on the existence of a «`Derivation`» between two requirements. When we launch the verification of this generic model, the profile identifies a violation of a constraint as shown in Figure 5.18.b. This violation is due to a missing rationale for the derived requirement. The rationale is required by activity 5.1.2.h for all software levels.

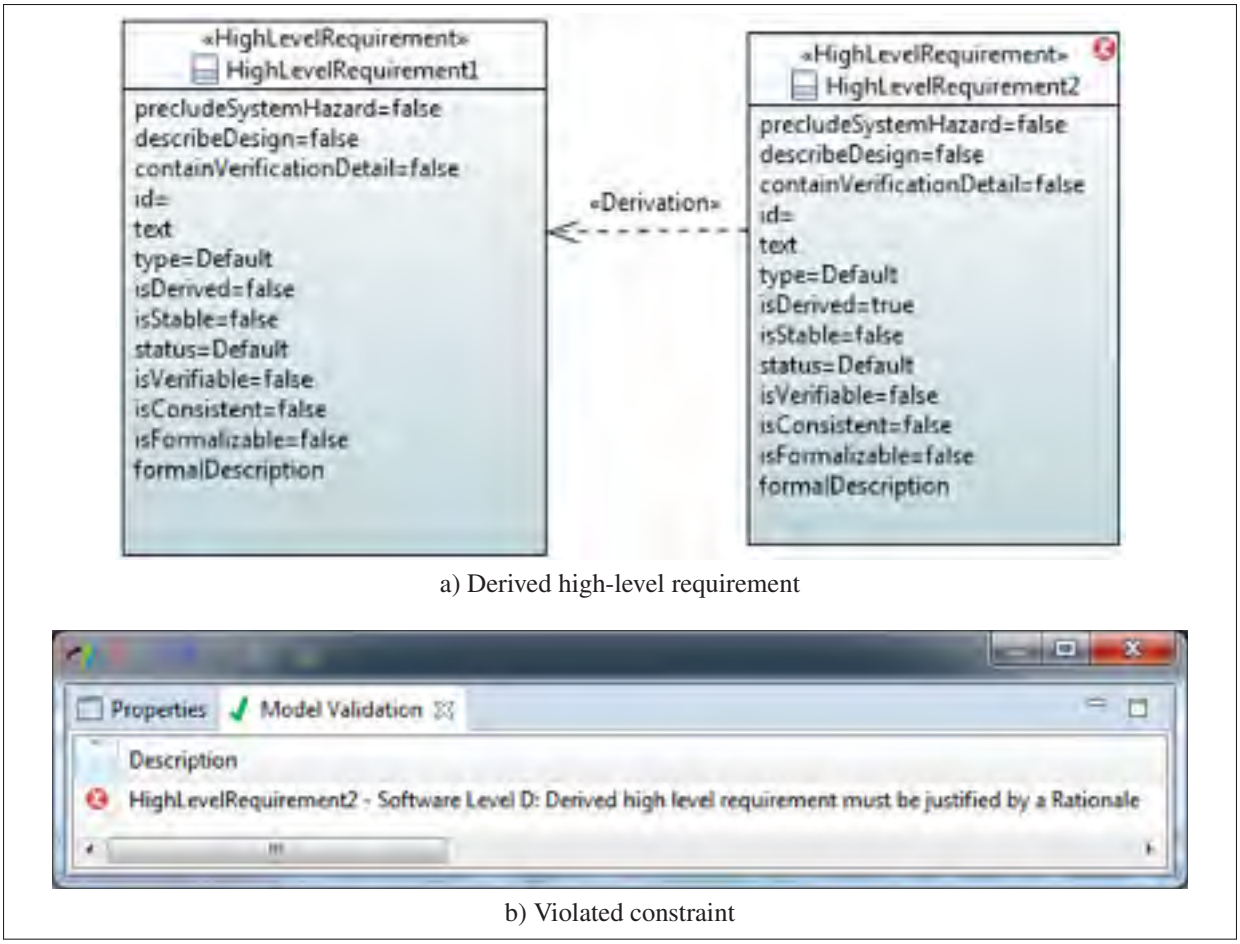


Figure 5.18 A derived high-level requirement violating one of the objectives of the standard.

Finnaly, by using the proposed profile, low-level requirements can be specified by either using the «LowLevelRequirement» stereotype (textual specification) as shown in Figure 5.19 or as the profile extends UML, low-level requirements can be specified using UML design diagram. As such, multiple low-level requirements may be specified by a single UML design diagrams. Figure 5.20 shows an example of UML state machine that implements to the behavior of multiple low-level requirements, including the low-level requirement specified in Figure 5.19 (LLR-44).

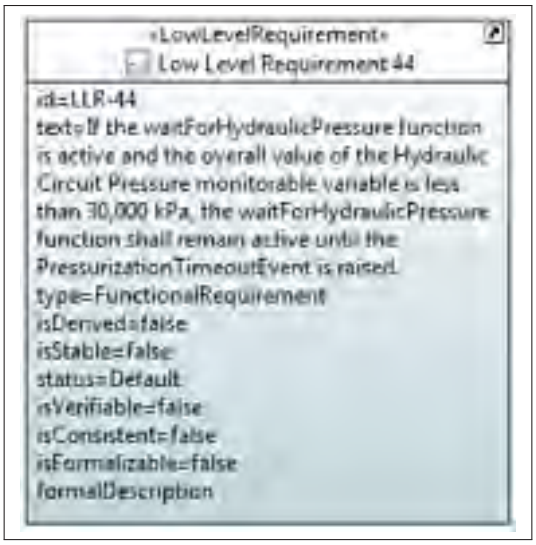


Figure 5.19 Low-level requirement 44.

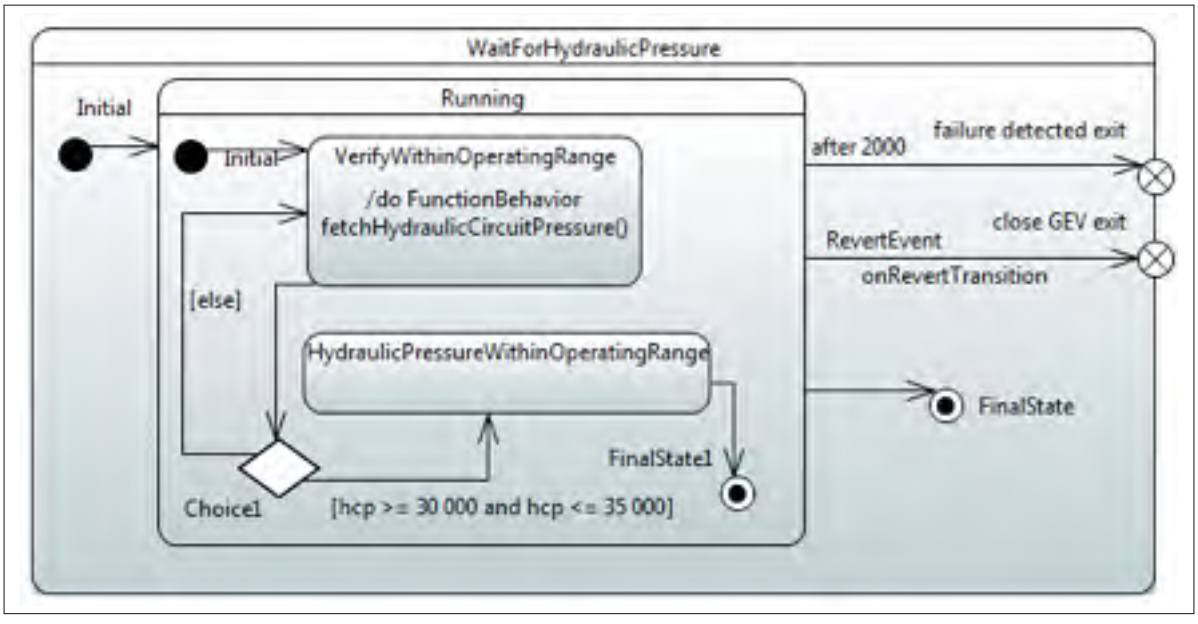


Figure 5.20 The WaitForHydraulicPressure state machine.

5.2.4 Use case 3: Ensuring traceability of software requirements

The goal of this use case is to demonstrate how the profile supports traceability between software requirements and software design. In the context of DO-178C, software design comprises software architecture and low-level requirements. In the following, we first present the software

architecture of the LGCS. Then we discuss the traceability between high-level requirements and software architecture, and between high-level requirements and low-level requirements.

Software architecture

The architecture of the LGCS is shown on Figure 5.21 as a UML component diagram. The components that constitute this architecture have been derived from the high-level requirements that define the behavior of the LGCS. The architectural design of the LGCS is based on the process control architectural style in which the system is driven by a set of received inputs that are used to determine the set of outputs that define the new state of the system.

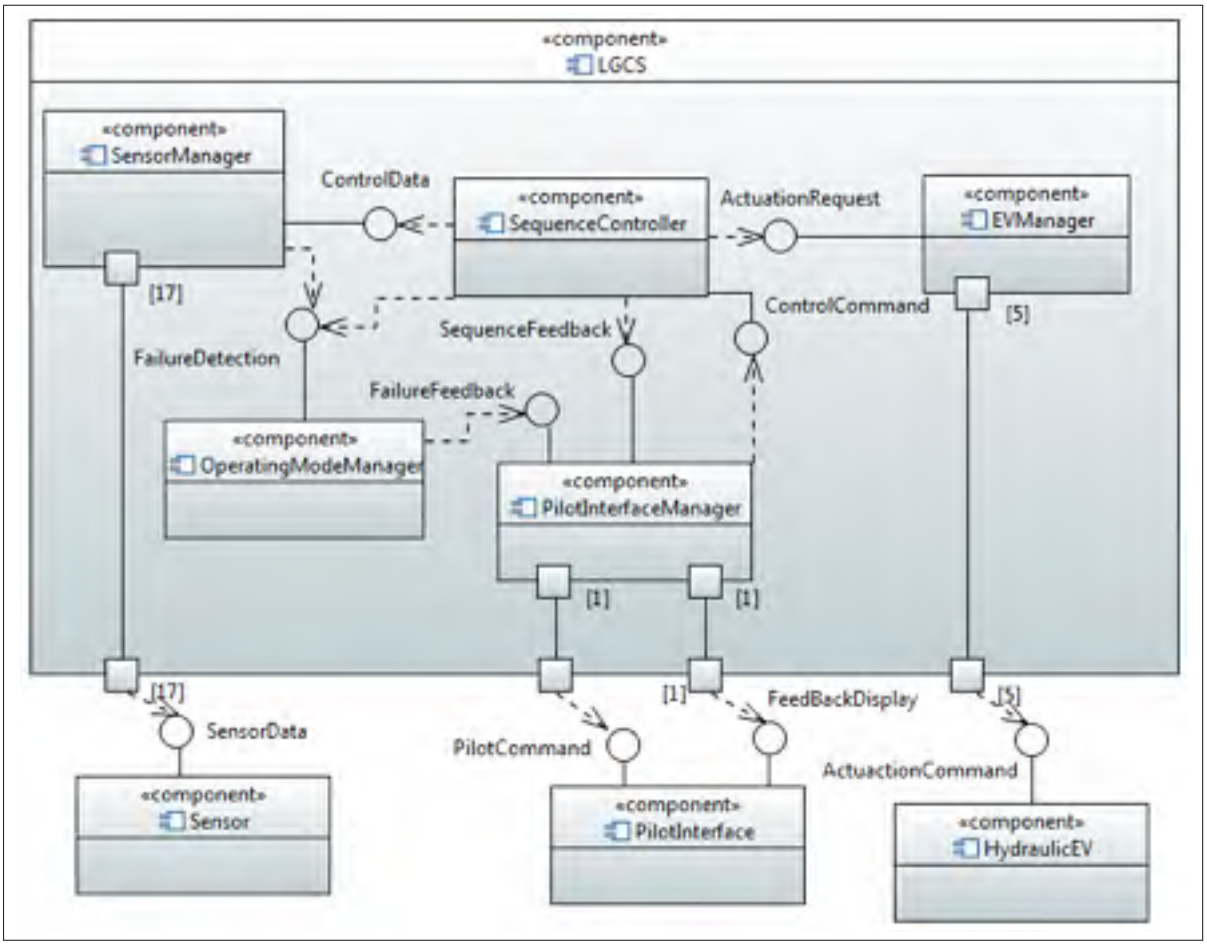


Figure 5.21 The landing gear control software architecture.

The LGCS receives a set of 17 inputs from the sensors that are attached to the physical entities that interface with the LGCS. These sensors are in charge of monitoring different part of the system. The `SensorManager` component is in charge of receiving the inputs from these sensors. The `SequenceController` component is in charge of performing the motion sequence of the landing gears that is requested by the pilot and received from the external `PilotInterface` component. Feedback about the overall system state is also provided to the pilot through the `PilotInterfaceManager` component that communicates with the `PilotInterface` component. Upon reception of the pilot request, the `SequenceController` interacts with the `SensorManager` to get the sensors readings. The `SensorManager` retrieves the values of the sensors and performs a validation of the obtained values in order to determine the validity of the received data and then transmit the overall requested values to the `SequenceController`. The `SensorManager` component also monitors failures that might be occurring in the system and reports those failures to the `OperatingModeManager` component. Finally, the `SequenceController` sends commands to the `EVManager` component which in turn sends commands to the external `HydraulicEV` component in order to activate the system's electro-valves allowing actuation of the landing gears.

Specifying traceability between the high-level requirements and the software architecture

Using the proposed profile, we enable the traceability of the high-level requirement and the software architecture by using the «`Satisfaction`» stereotype. This relationship is used to specify the traces between the requirements and the entities of the architecture or the design that are responsible for the implementation of the requirements.

Figure 5.22 provides a view of the LGCS architecture as created using the profile. This figure shows explicitly the traces between the subset of the LGCS high-level requirements provided in Table 5.1 and the software components to which they are apportioned. As observed, HLR-1 is implemented by the `SensorManager` component while HLR-4, HLR-6, HLR-7 and HLR-12

are allocated to the `SequenceController` component. HLR-4 is also allocated to the `PilotInterfaceManager` component.

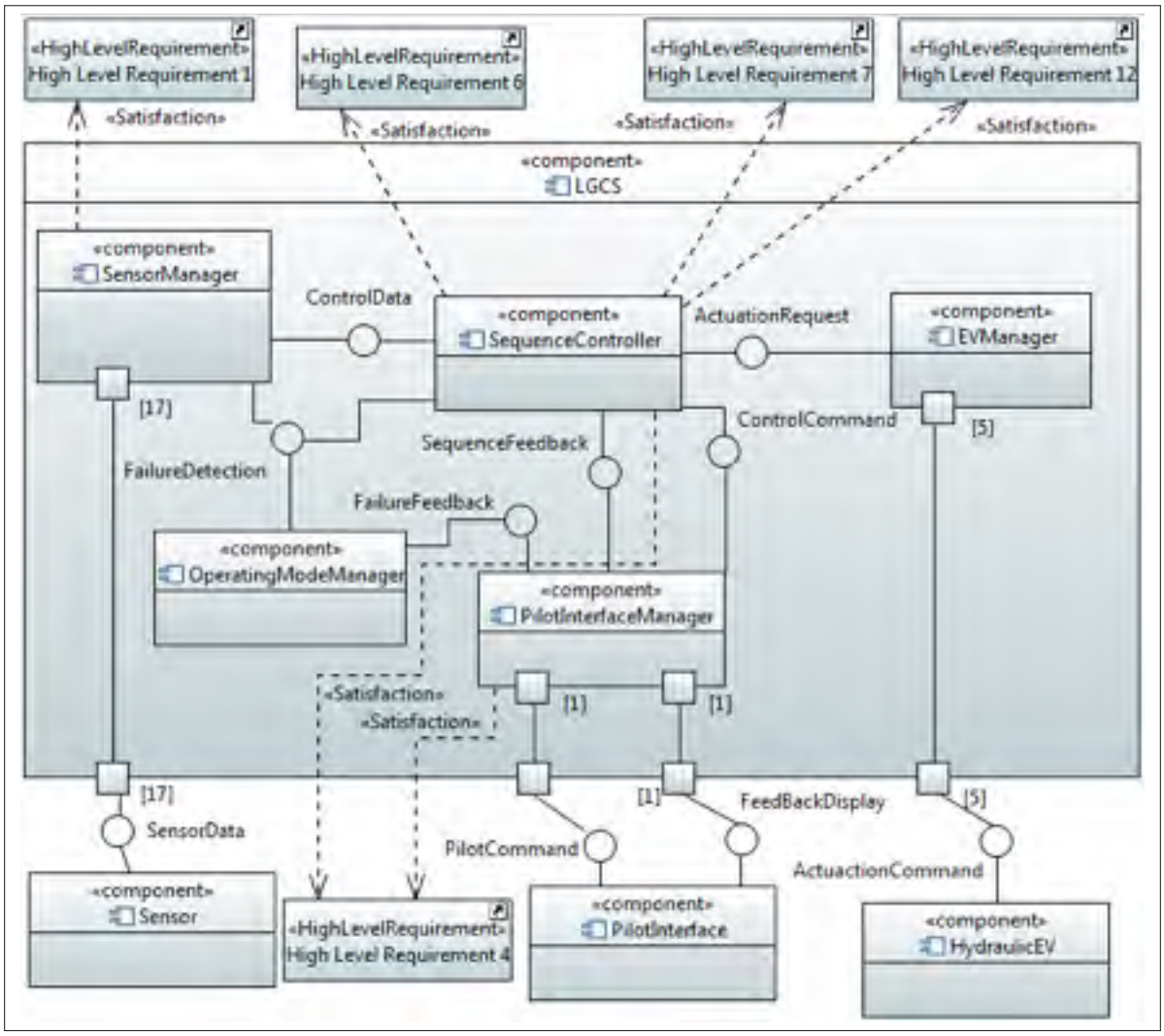


Figure 5.22 A subset of the LGCS HLRs and their related components.

Figure 5.23 shows another view, created using our profile, in which a subset of the software architecture is depicted. The `SequenceController` component and the high-level requirements it implements are shown along with the class that realizes the `SequenceController` component.

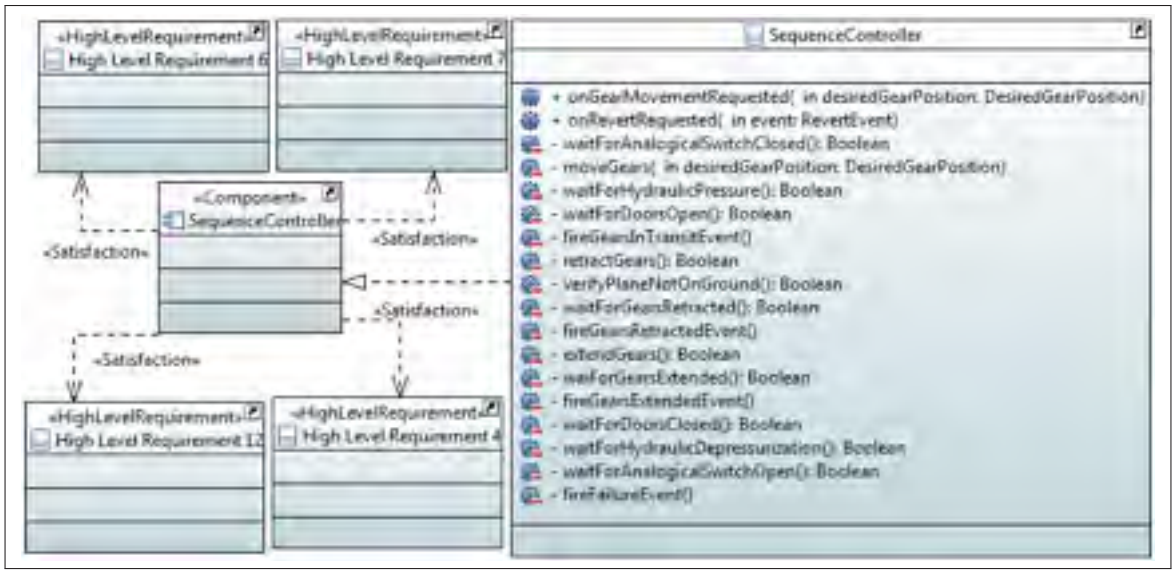


Figure 5.23 The SequenceController tracing to the high-level requirements it satisfies along with its realizing class.

Specifying traceability from the high-level requirements to the low-level requirements

Using the DO-178C profile, high-level requirements and low-level requirements are specified in a similar manner. However HLRs are specified by the «`HighLevelRequirement`» stereotype and LLRs are specified by the «`LowLevelRequirement`» stereotype. Traceability between software requirements is achieved by using the «`Refinement`», the «`Derivation`», and the «`Satisfaction`» stereotypes.

As our profile is integrated within a UML modeling environment (i.e. Papyrus), LLRs can be specified using both UML design diagrams (e.g. state machines) and textual descriptions. Both specifications can be traced to HLRs using the profile.

Consider again the LLRs specified by the `waitForHydraulicPressure` state machine presented in Figure 5.20. Because Papyrus does not allow us to add any profile stereotype to UML state machine diagrams, we used class diagrams to create traceability links between LLRs and HLRs using our profile when LLRs are specified as state machines.

Figure 5.24 provides a view of the traceability between HLR-4 and the elements of the `waitForHydraulicPressure` state machine that implement HLR-4. It is implemented by the transition that raises the `RevertEvent` that goes to the `CloseGEVExit`. Figure 5.25 shows the elements of the `waitForHydraulicPressure` state machine that trace to HLR-6. Specifically HLR-6 is implemented by a subset of elements of the `waitForHydraulic-Pressure` state machine. It is implemented by the `Running` state that reads the hydraulic pressure and checks if it is within the operating range.

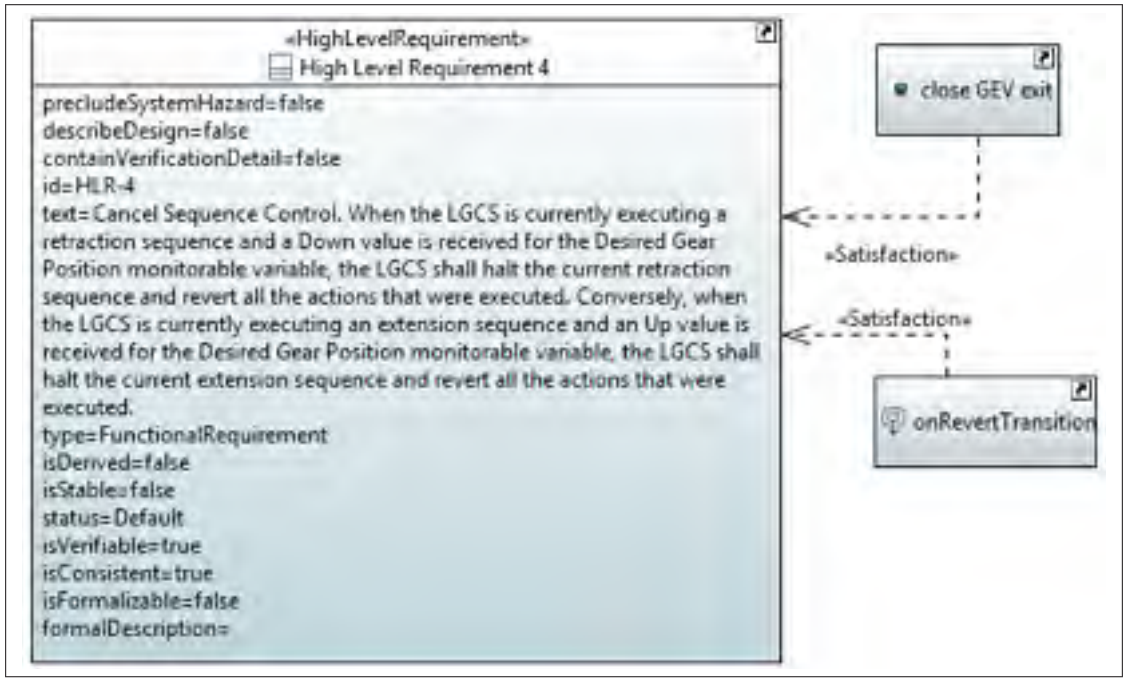


Figure 5.24 Elements of the WaitForHydraulicPressure state machine that trace to HLR-4

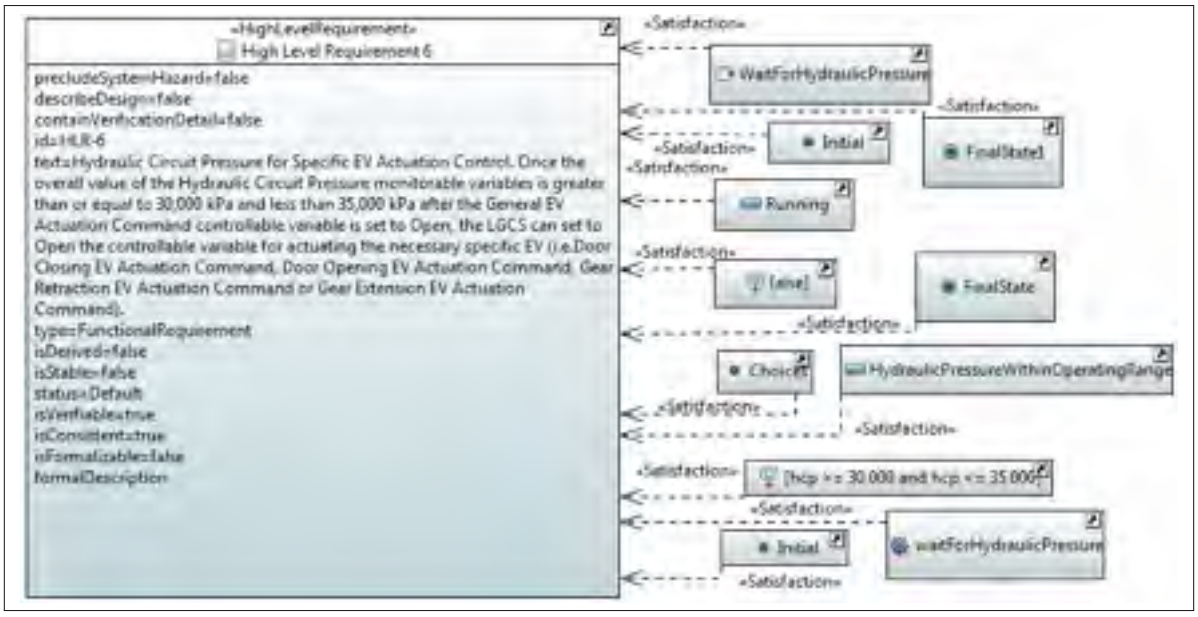


Figure 5.25 Elements of the WaitForHydraulicPressure state machine that trace to HLR-6.

Figure 5.26 shows an example of LLRs where the LLR (LLR-44) has been specified using the «`LowLevelRequirement`» stereotype introduced by the profile. In this case the traceability to HLRs is ensured through the «`Refinement`» relationship. In addition to tracing LLRs to HLRs, the profile enables to trace LLRs specified using the «`LowLevelRequirement`» stereotype to LLRs specified by UML design diagrams.

Figure 5.27 shows the traceability of LLR-44 to the design elements that are responsible for its implementation. LLR-44 is implemented by the `waitForHydraulicPressure` operation. The complete behavior of the function is captured by the `waitForHydraulicPressure` state machine, however only a subset of elements of the state machine actually implements LLR-44. These elements are the `Running` and the `VerifyWithinOperatingRange` states, the transition named `else` that results from the choice present in the `Running` state.

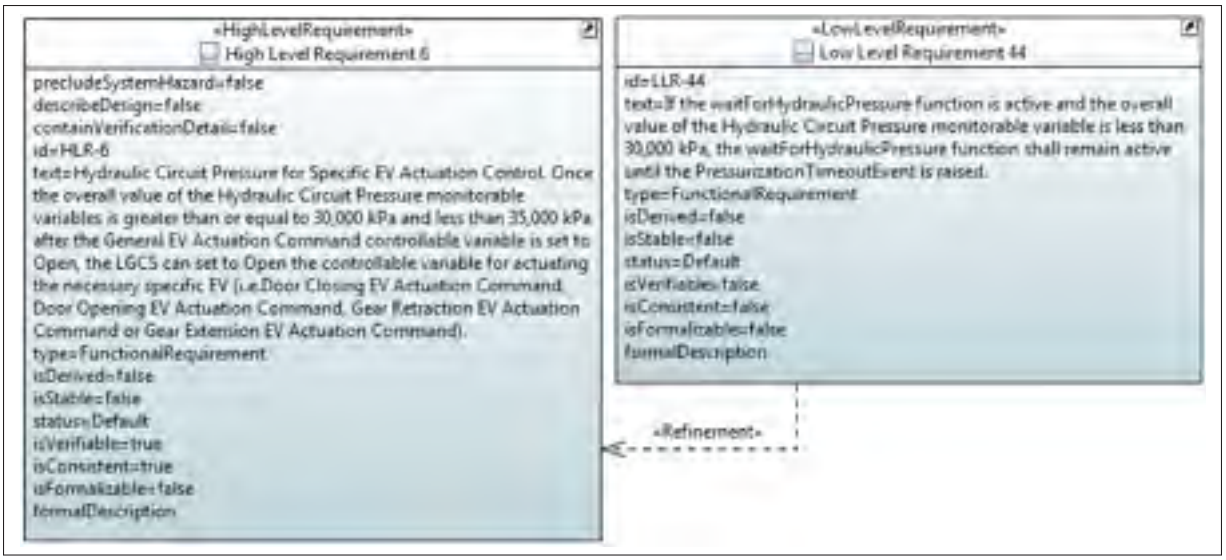


Figure 5.26 Refinement of HLR-6 into LLR-44.

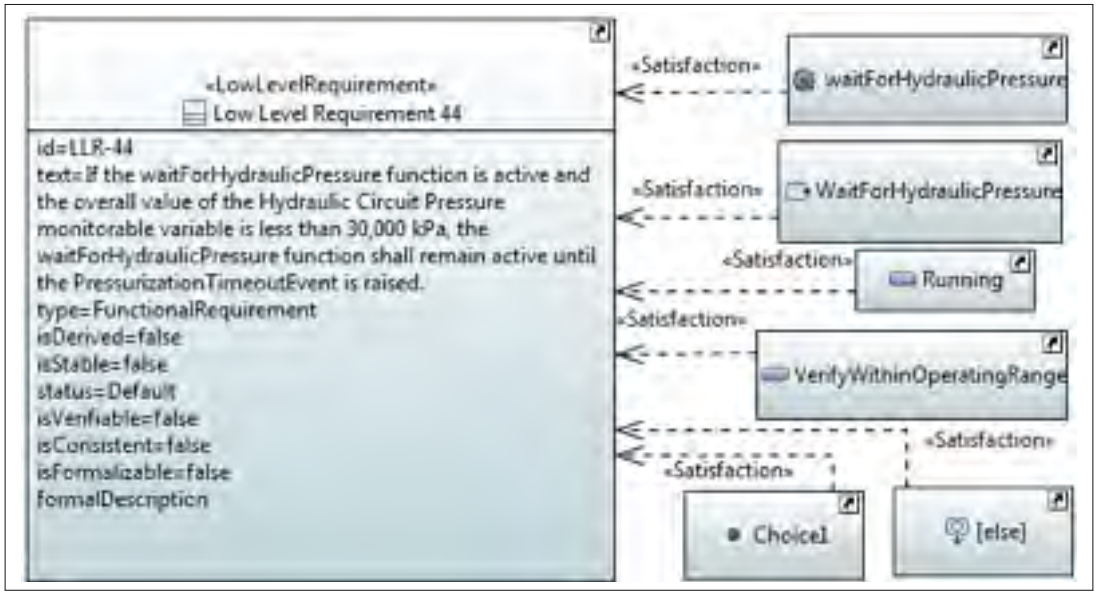


Figure 5.27 Elements of the waitForHydraulicPressure that statisfy LLR-44.

5.2.5 Use case 4: Specifying verification data

The activities of the software verification process aim at assessing the outputs of the software planning process, software development processes, and the software verification process.

The software verification process produces data that are a combination of reviews, analyses, test cases and procedures. The proposed profile enables the specification of these data and their traceability. Specifically, we focus on the traceability of verification data to software requirements and design. This is done using the verification diagram and package defined in the profile. The remaining of this section will provide examples of testing data that we were able to produce within the scope of our work.

Consider the HLR-12 shown in Figure 5.28. HLR-12 states that once 2 seconds have elapsed since the general EV Actuation Command controllable variable was set to Open and the overall value of the Hydraulic Circuit Pressure monitorable variables is still less than 30,000 kPa, the LGCS shall detect a failure of the general hydraulic electro-valve. Figure 5.29 shows the specification of a normal range test case that verifies that HLR-12 is satisfied. This specification describes the test purpose, pass and fail criterion and its expected result. In order for the test to be successful, the LGCS shall detect a failure of the system when the pressure remains below 30 kPa once 2 seconds have elapsed since the general electro valve have been set to open.

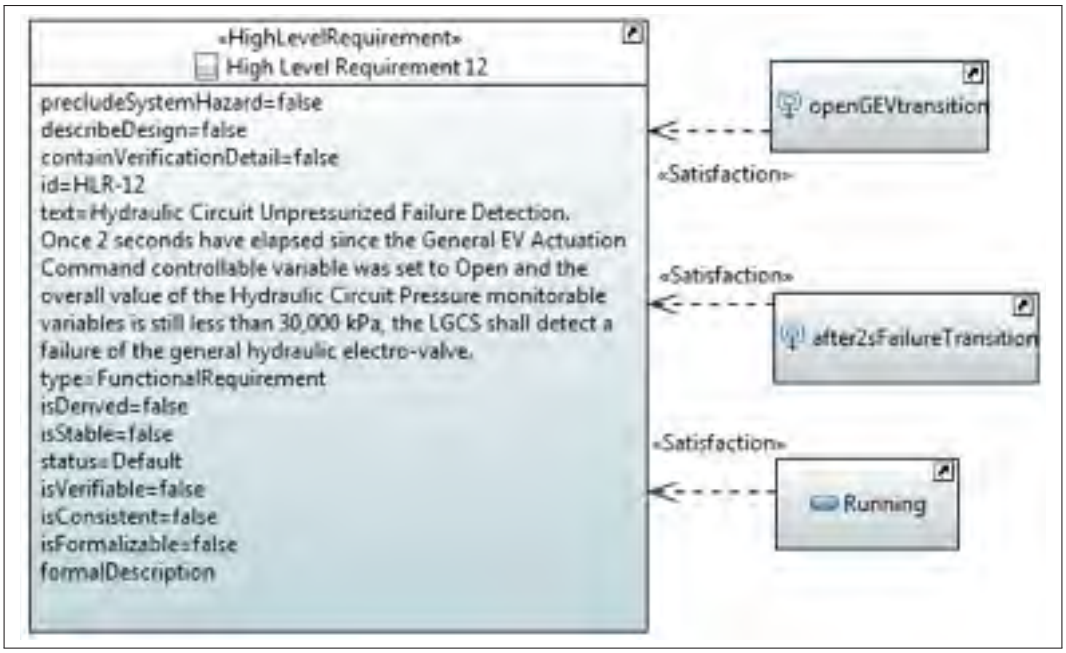


Figure 5.28 An example of high-level requirement.

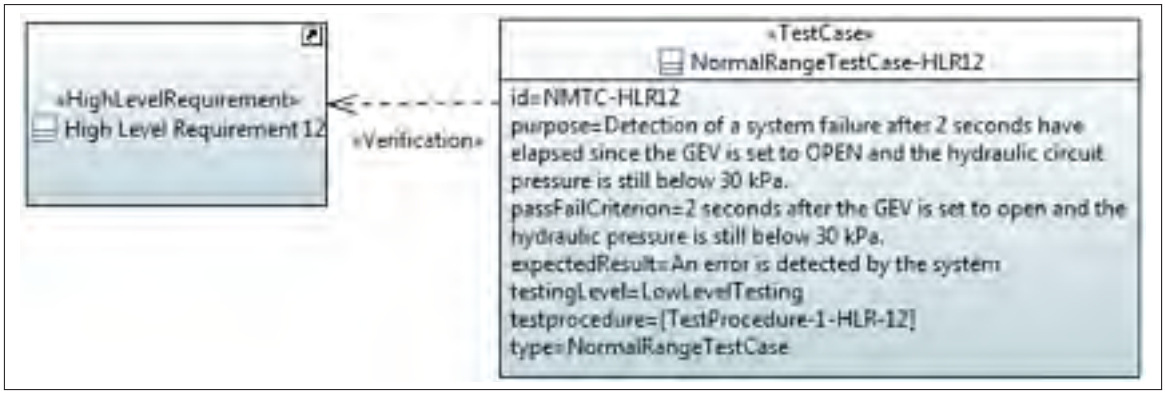


Figure 5.29 Specification of a normal range test case intended to verify the correct behavior of the LGCS as specified in HLR-12.

Figure 5.30 depicts a test procedure for executing the test case provided in Figure 5.29. This procedure describes the environment in which the testing occurs. In this case, the test case shall be executed in a black box environment. The LGCS is running in its normal mode of operation (i.e. the system is not in a failed mode) and the received (simulated) pressure of the hydraulic circuit shall never be superieur or equal to 30 kPa. The LGCS shall receive a new desired gear

position input value through the pilot interface manager. Because we did not execute the LGCS test cases, we do not have any test results. This is illustrated in Figure 5.30 by the fact that `testResult` is null.

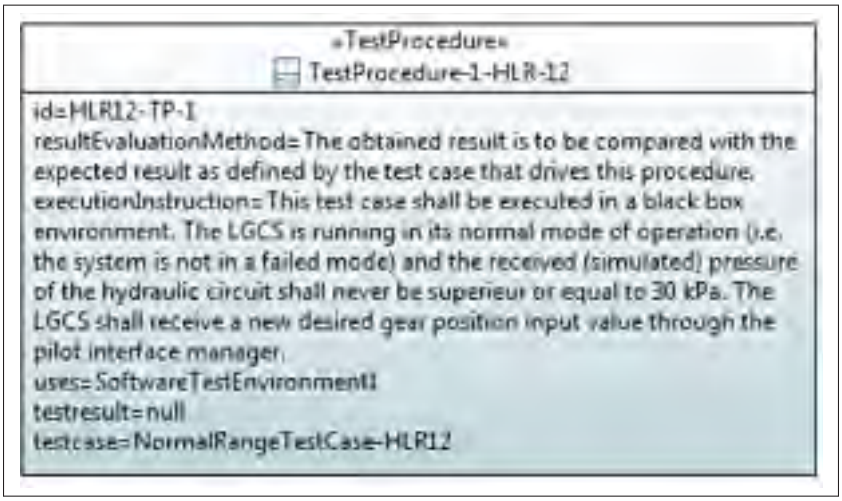


Figure 5.30 Test procedure associated to the normal range test case provided in Figure 5.29.

Figure 5.31 provides the specification of a review that targets LLR-44 following objective 6.3.2.a. The objective is to verify that low-level requirements comply with high-level requirements, and that a rationale is provided for each derived LLR. In this case, LLR-44 is not a derived requirement so the review does not need to check that a rationale is provided for this LLR. In order to perform this review, no additional testing data is required as the review analyses the compliance between the LLR and the HLRs being refined. The result of the review is that the behavior specified by LLR-44 complies with the high-level requirements it refines (HLR-6 and HLR-12).

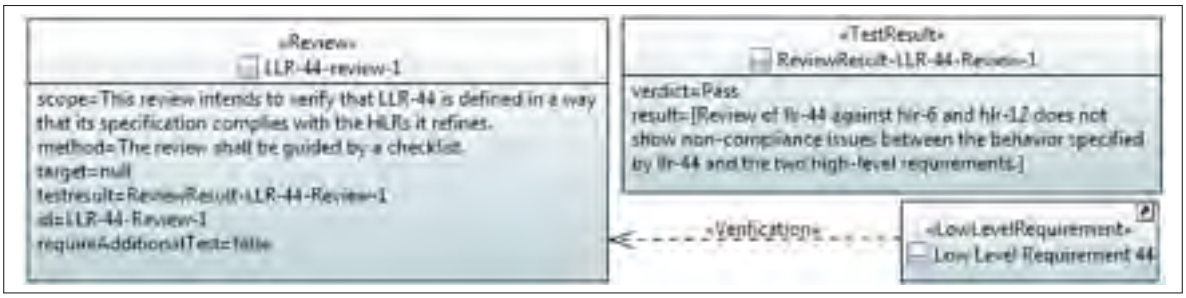


Figure 5.31 Specification of a review as defined by objective 6.3.2.a along with the result of the conducted review.

5.3 Discussion

We used the DO-178C profile to model the LGCS case study. We were able to specify all the high-level and low-level requirements. We also used the profile to model the software architecture and low-level requirements in the form of standard UML diagrams. Using the profile, we were also able to specify verification data. Finally, the profile helped specifying the traceability links between HLRs and LLRs, between HLRs and software architecture. Moreover the profile enables to trace the specification of LLRs to their specification as UML standard design diagrams.

Nevertheless, we could not demonstrate some of the features of our profile because of the limits of the LGCS case study. In fact, the LGCS specification does not describe test cases nor test procedures and results. So to illustrate the usefulness of the profile, we have defined test cases and procedures ourselves. As a result, we could not demonstrate the use of certain concepts of the profile.

The usefulness of the proposed profile depends on the extent to which it complies with/and supports DO-178C guidelines. In this context, our conceptual model of DO-178C was validated by industrial partners through a number of working sessions. The profile will be deployed in an industrial context in the near future.

The usability of our profile may also be limited by the modeling environment (Papyrus) that we used to implement it. During our work with Papyrus, we realized that some UML concepts are not supported. For instance Papyrus does not enable to specify the UML elements "signal reception" and "receive signal" that are used in both state machine and activity diagram.

CONCLUSION AND FUTURE WORKS

Contributions

In this thesis, we proposed a model-driven approach to support the process of collecting evidence required for software certification in the context of airborne systems that must be compliant with DO-178C. In particular, we built a domain-specific modeling language that supports the specification of such evidence. Thus this language provides modeling constructs that match DO-178C concepts and vocabulary. We implemented our language as a UML profile (DO178C profile) using an open-source modeling tool (Papyrus).

Our DO178C profile has the following unique features:

- The profile provides means to specify the software life cycle data in terms of planning models and integrates these models with those produced by the software development and verification processes.
- The profile provides means to specify the objectives and activities to be performed throughout the software life cycle depending on the targeted assurance level and applied DO-178C supplements. This is enforced through a number of constraints that apply to the planning models being created by the user. Each constraint is defined for a specific assurance level and may result from a specific DO-178C supplement. Moreover, the profile provides templates that generate automatically objectives and activities depending on the assurance level chosen by the user.
- The profile provides means for modeling software requirements and verification data as required by DO-178C. As the profile extends the UML meta-model, software requirements and verification data is integrated with design models. Furthermore, the profile supports the

traceability between requirements and verification data, and between high-level and low level requirements.

We performed a case study to assess our profile. Specifically, we used the profile to model a realistic example of airborne software (i.e. the LGCS). We illustrated its usage through four particular use cases. These use cases helped demonstrate the usability and usefulness of the profile. However the demonstration of the features of the profile was limited to those required by the LGCS.

Future works

In the near future, we plan to refine our profile and extend our work as follows:

- **Refining the profile through more case studies:** First we plan to use our profile to model another available airborne software (i.e. The Helicopter Flight Control System (Mathworks, 2017)). Second, we plan to deploy the profile in an industrial context. This will help refine the modeling constructs and constraints defined by the profile.
- **Implementing and integrating the profile within a commercial tool:** Our current implementation of the profile is based on the Papyrus open-source tool. Papyrus does not support some UML concepts which impacts the usability and usefulness of our profile. Since our industrial partners are using the IBM modeling suite, we plan to implement the profile within the Rational Rhapsody tool.
- **Automating the generation of (part of) the documentation required for certification:** Since our profile provides means to specify software life cycle data, our work can be extended to automatically generate documents from these data. Indeed, to get certified, an applicant must submit such documents to the certifying authorities. Currently, these

documents are still manually written and a large effort is required to collect all of the required information which result from different processes and activities. We believe that our profile can be used to generate, if not entirely, sections of these documents. To do so, we plan to explore existing model query engines such as VIATRA (Viatra, 2017).

APPENDIX I

OBJECTIVES AND ACTIVITIES OF THE SOFTWARE PLANNING PROCESS

	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	Ref	Ref	A	B	C	D	Data Item	Ref	A	B	C	D
1	The activities of the software life cycle processes are defined.	4.1.a	4.2.a 4.2.c 4.2.d 4.2.e 4.2.g 4.2.i 4.2.l 4.3.c	O	O	O	O	PSAC SDP SVP SCM Plan SQA Plan	11.1 11.2 11.3 11.4 11.5	① ① ② ②	① ① ② ②	① ② ② ②	① ② ② ②
2	The software life cycle(s), including the inter-relationships between the processes, their sequencing, feedback mechanisms, and transition criteria, is defined.	4.1.b	4.2i 4.3.b	O	O	O		PSAC SDP SVP SCM Plan SQA Plan	11.1 11.2 11.3 11.4 11.5	① ① ① ①	① ① ① ①	① ② ② ②	
3	Software life cycle environment is selected and defined.	4.1.c	4.4.1 4.4.2.a 4.4.2.b 4.4.2.c 4.4.3	O	O	O		PSAC SDP SVP SCM Plan SQA Plan	11.1 11.2 11.3 11.4 11.5	① ① ① ①	① ① ① ①	① ② ② ②	
4	Additional considerations are addressed.	4.1.d	4.2.f 4.2.h 4.2.i 4.2.j 4.2.k	O	O	O	O	PSAC SDP SVP SCM Plan SQA Plan	11.1 11.2 11.3 11.4 11.5	① ② ① ② ①	① ② ① ② ①	① ② ② ② ②	① ② ② ② ②
5	Software development standards are defined.	4.1.e	4.2.b 4.2.g 4.5	O	O	O		SW Requirements Standards SW Design Standards SW Code Standards	11.6 11.7 11.8	① ① ①	① ① ①	② ② ②	
6	Software plans comply with this document.	4.1.f	4.3.a 4.6	O	O	O		Software Verification Results	11.14	②	②	②	
7	Development and revision of software plans are coordinated.	4.1.g	4.2.g 4.6	O	O	O		Software Verification Results	11.14	②	②	②	

Figure-A I-1 Objectives and activities of the Software Planning Process. Extracted from RTCA (2011a).

APPENDIX II

THE TYPE PACKAGE OF THE DO-178C PROFILE

The UML profile for DO-178C introduces types that are used to specify the values of the attributes of the stereotypes. The Types package contains the definition of each of these types. The following sections provide the description of each of these types and their possible values.

1. ActivityState

Enumeration Name	Description	Values
ActivityState	Specifies the possible values for an activity state.	• InProgress • Pending • Terminated • UnderReview • UnderCorrection

2. ActivityType

Enumeration Name	Description	Values
ActivityType	Provides the list of all activities defined by the standard.	For the complete list of activities refer to the standard.

3. ControlCategory

Enumeration Name	Description	Values
ControlCategory	Control categories are configuration management controls that are placed on the software life cycle data. There exist two categories that define the activities of the software configuration management process to be applied to software life cycle data.	• ControlCategory1 • ControlCategory2

4. DO178CSupplement

Enumeration Name	Description	Values
DO178CSupplement	Supplements represent the set of documents that are to be used together with DO-178C guidelines when specific technologies are used.	• DO330: Represents the DO-330 Software Tool Qualification Considerations supplement. • DO331: Represents the DO-331 Model-Based Development and Verification Supplement to DO-178C and DO-278A. • DO332: Represents the DO-332 Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A. • DO333: Represents the DO-333 Formal Methods Supplement to DO-178C and DO-278A.

5. ObjectiveType

Enumeration Name	Description	Values
ObjectiveType	Provides the list of all of the objectives defined by the standard.	For the complete list of objectives refer to the standard.

6. ProcessType

Enumeration Name	Description	Values
ProcessType	Provides the list of DO-178C defined processes that constitute the software life cycle.	• SoftwarePlanningProcess • SoftwareRequirementsProcess • SoftwareDesignProcess • SoftwareCodingProcess • IntegrationProcess • SoftwareVerificationProcess • SoftwareConfigurationManagementProcess • SoftwareQualityAssuranceProcess • CertificationLiaisonProcess

7. RationaleType

Enumeration Name	Description	Values
RationaleType	This is used to specify the type of a rationale for specific use cases as required by DO-178C.	• Undefined: A rationale without type. • DesignDetailJustification: A rationale used to justify why a high-level requirement contains design details. • VerificationDetailJustification: A rationale used to justify why a high-level requirement contains verification details.

8. RequirementStatus

Enumeration Name	Description	Values
RequirementState	Defines the possible values for the status of a requirement.	• Unreviewed • Reviewed&Incorrect • Reviewed&Accepted

9. RequirementType

Enumeration Name	Description	Values
RequirementType	This is used to identify the kind of the requirement being specified.	• FunctionalRequirement • OperationalRequirement • InterfaceRequirement • PerformanceRequirement • SecurityRequirement • MaintenanceRequirement • CertificationRequirement • AdditionalRequirement • SafetyRelatedRequirement • ComponentResuseRequirement

10. SoftwareLevel

Enumeration Name	Description	Values
SoftwareLevel	This is used to specify the criticality level of the software to be produced.	• SoftwareLevelA • SoftwareLevelB • SoftwareLevelC • SoftwareLevelD • SoftwareLevelE

11. SoftwareLifeCycleDataType

Enumeration Name	Description	Values
SoftwareLifeCycleDataType	This is used to specify the kind of data that a software life cycle data represents.	• PlanForSoftwareAspectsOfCertification • SoftwareDevelopmentPlan • SoftwareVerificationPlan • SoftwareConfigurationManagementPlan • SoftwareQualityAssurancePlan • SoftwareRequirementsStandards • SoftwareDesignStandards • SoftwareCodeStandards • SoftwareRequirementsData • SoftwareDesignDescription • SourceCode • ExecutableObjectCode • SoftwareVerificationCasesAndProcedures • SoftwareVerificationResults • SoftwareLifeCycleEnvironmentConfigurationIndex • SoftwareConfigurationIndex • ProblemReports • SoftwareConfigurationManagementRecords

Enumeration Name	Description	Values
SoftwareLifeCycleDataType		• SoftwareQualityAssuranceRecords • SoftwareAccomplishmentSummary • TraceData • ParameterDataItemFile • SoftwareModelStandards • Feedback

12. TestingMethod

Enumeration Name	Description	Values
TestingMethod	Defines the kind of the test to be carried out.	• HardwareSoftwareIntegrationTesting • SoftwareIntegrationTesting • LowLevelTesting

13. TestTarget

Enumeration Name	Description	Values
TestTarget	Specifies the possible values for the targeted environment for the execution of a test case.	• TargetComputer • TargetComputerEmulator • HostComputerSimulator

14. TestType

Enumeration Name	Description	Values
TestType	Specifies the possible values for the type of a test case.	• NormalRangeTestCase • RobustnessTestCase

15. Verdict

Enumeration Name	Description	Values
Verdict	A verdict is used to specify the conclusions of evaluating the results obtained from the execution of a test. The verdict enumeration is consistent with the definition of a verdict as defined by the UML testing profile (UTP) (OMG, 2014).	• None: The test case, test procedure, review or analysis have not yet been executed. • Pass: The test results are consistent with the expected results. • Inconclusive: The evaluation of the test results is inconclusive, i.e we cannot state whether the test fails or passes. • Fail: The test results are not consistent with the expected results. • Error: There have been an error within the testing environment.

APPENDIX III

INTEGRATING THE DO-178C PROFILE WITHIN PAPYRUS

1. Identifying and registering the profile

The `org.eclipse.papyrus.do178c` plug-in is in charge of the registration of a static profile within Papyrus. Static profiles are generated from dynamic profiles (the profile model).

In order to build this plug-in, we have followed the instructions contained in the complete step by step guide provided in the papyrus user guide related to static profile generation [1].

This plug-in uses the following extension points provided by the Papyrus environment:

- *org.eclipse.emf.ecore.generated_package*: This extension point registers a generated Ecore package against a namespace URI within EMF's global package registry. This extension is automatically added to the plug-in and its content is filled for each of the profile's packages during the generation of the profile's source code from the generator model. Figure III-1 provides an example of the filled extension point DO178C package containing our profile definition.

- *org.eclipse.emf.ecore.uri_mapping* This extension point is used to define mappings that are to be applied by the environment's default URI converter when normalizing URIs. In simpler words, this extension point is used to define aliases for file paths which eases paths manipulation. Figure III-2 provides an example of how we have defined the URI mapping for plug-ins.

- *org.eclipse.uml2.uml.generated_package*: This extension point registers the location of a UML package against the namespace URI of its generated Ecore representation. The location attribute requires the *ID* of the profile that is found within the `profile.uml` file.

[1] "Generating Static Profiles": http://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.papyrus.uml.diagram.profile.doc%2Ftarget%2Fgenerated-eclipse-help%2Fusers%2FgeneratingStaticProfiles.html

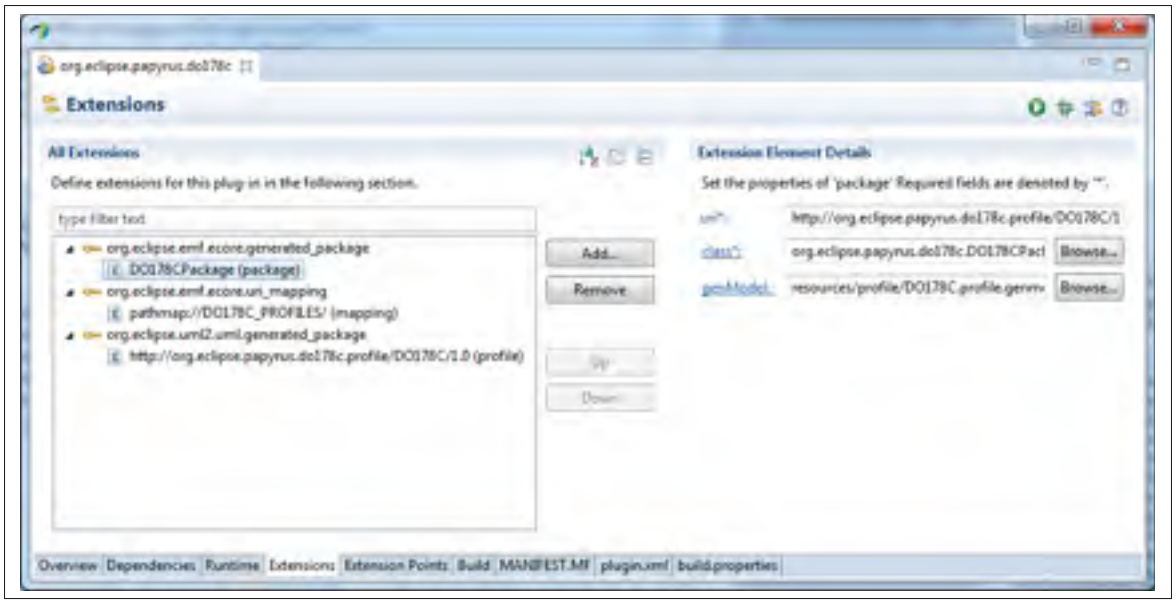


Figure-A III-1 DO178C generated package registration

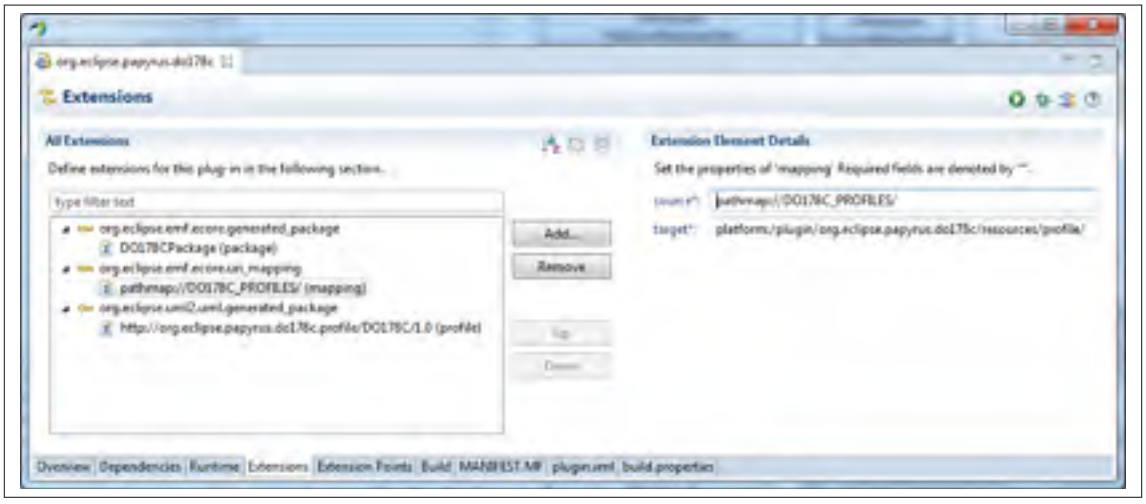


Figure-A III-2 DO178C profile URI mapping.

The *ID* location in the file is shown on Figure III-3. Figure III-4 provides an example of how we have filled the required inputs for the extension.

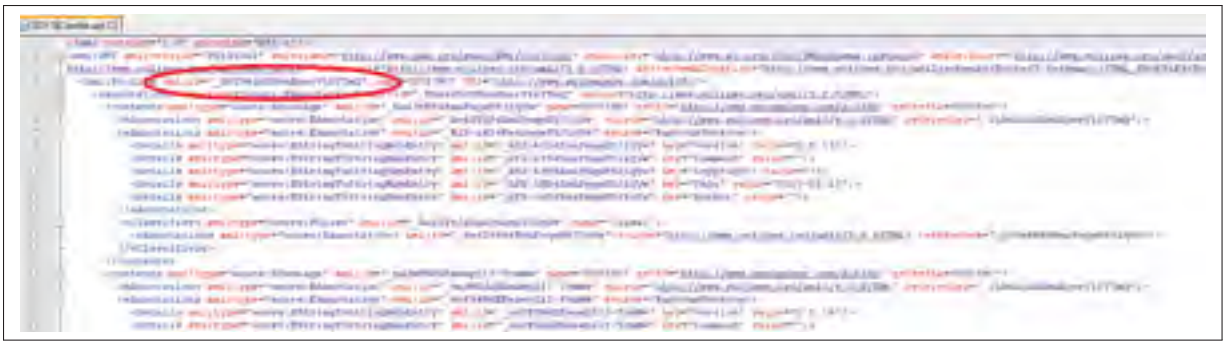


Figure-A III-3 The profile ID within the *profile.uml* file.

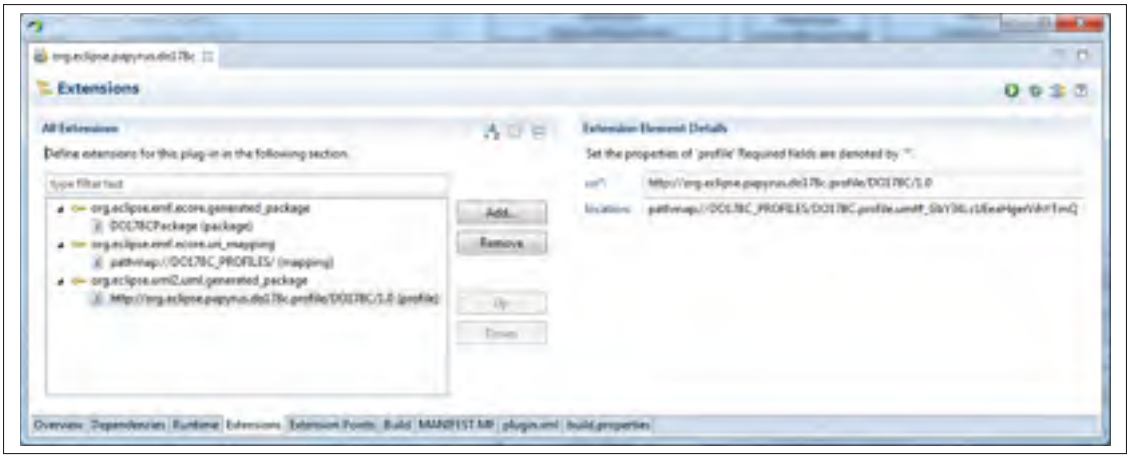


Figure-A III-4 Generated package registration.

2. Extending the Papyrus UI

The `org.eclipse.papyrus.do178c.ui` plug-in is in charge of the definition of multiple contributions to the user interface. These elements are defined through the following extension points:

- *org.eclipse.papyrus.uml.extensionpoints.UMLProfile*: This extension point registers UML profiles packaged as plug-ins into the Papyrus modeling tool. Complete guidance to fill-in the information related to this extension is provided in the papyrus user guide related to static profile generation[2]. Figure III-5 shows an example of the filled data for the registration of our profile.

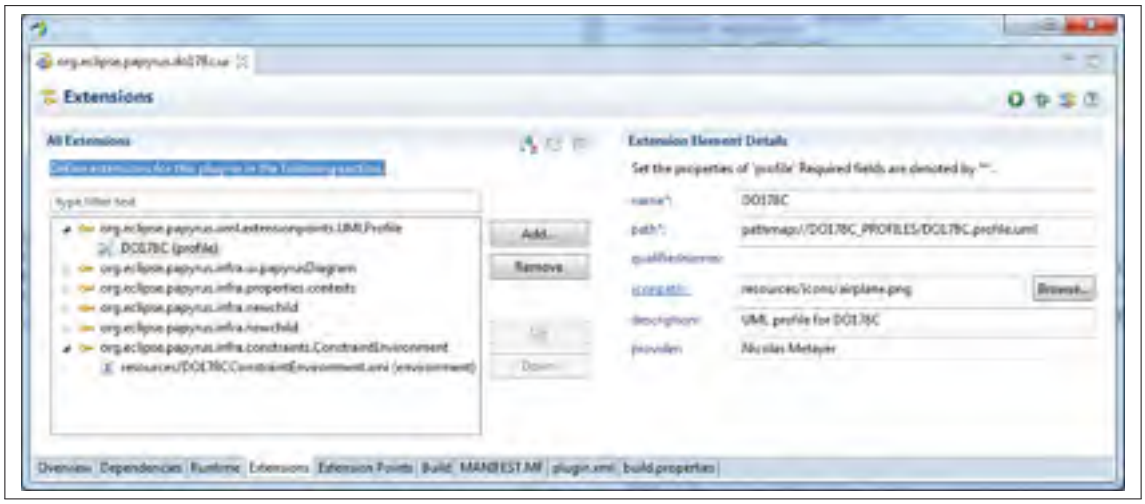


Figure-A III-5 UML profile registration.

- *org.eclipse.papyrus.infra.ui.papyrusDiagram* This extension point is used to register new diagram editors within Papyrus. Papyrus uses the term diagram category to categorize domain models. Papyrus supports UML and UML Profiles as default domZain models. Papyrus also uses the term diagram kind to refer to the specialization of the diagram editor

[2] "Generating Static Profiles": http://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.papyrus.uml.diagram.profile.doc%2Ftarget%2Fgenerated-eclipse-help%2Fusers%2FgeneratingStaticProfiles.html

for a certain domain, for example the UML class diagram is a diagram kind for the UML domain category. A diagram kind belongs to only one diagram category whereas a diagram category may contain multiple diagram kinds. Figures III-6 and III-7 respectively show the diagram category page and the diagram kind page when creating a new model within Papyrus. Figure III-8 shows how we have filled the extension for our DO178C diagram category.

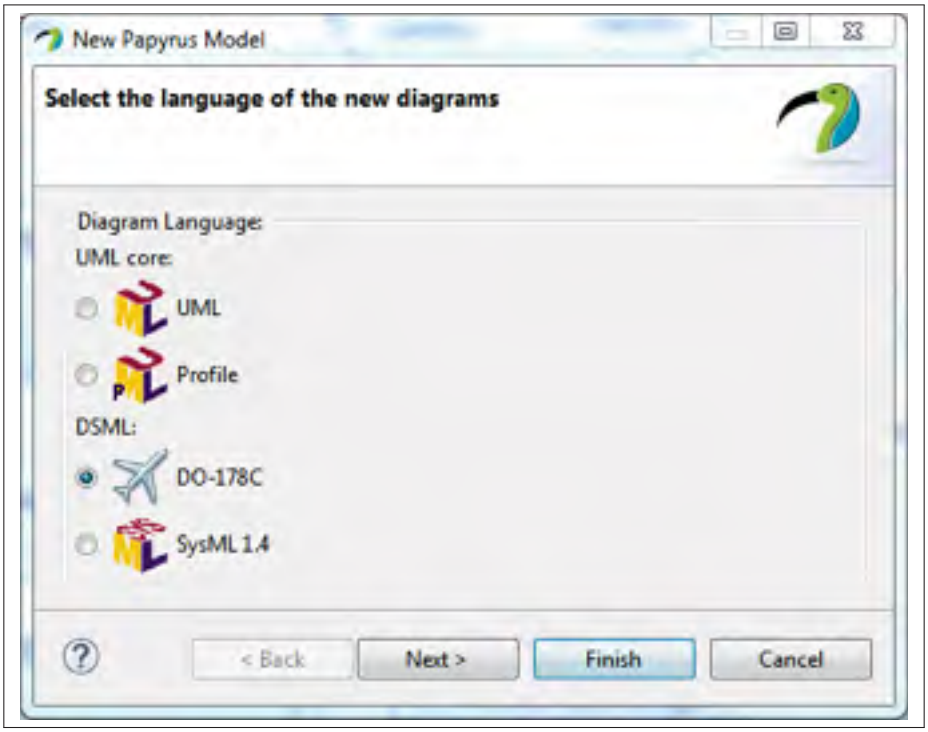


Figure-A III-6 Papyrus diagram category selection wizard.

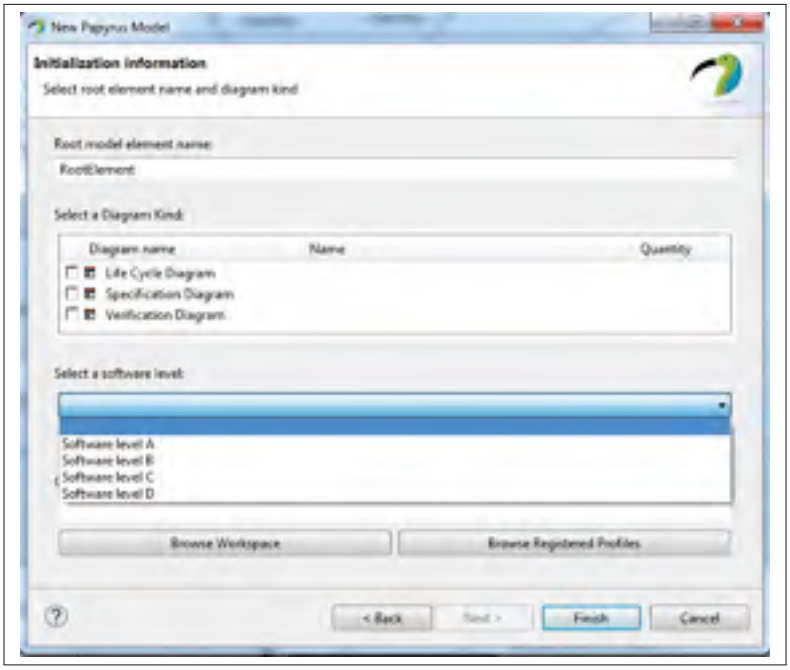


Figure-A III-7 Papyrus diagram kind selection wizard.

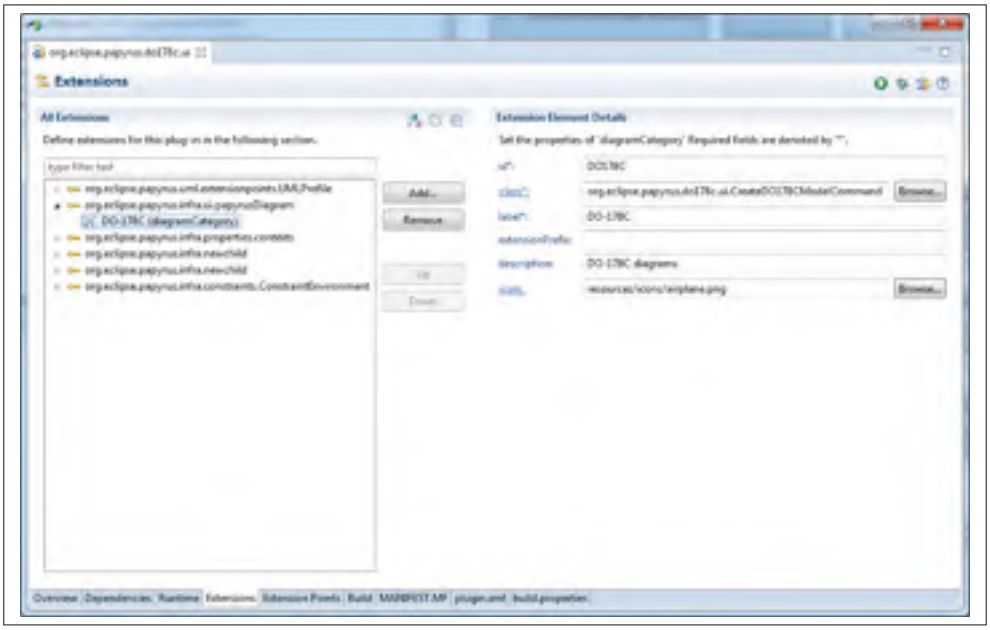


Figure-A III-8 Diagram category registration.

- *org.eclipse.papyrus.infra.properties.contexts*: This extension point allows the registration of property view pages for the stereotypes of the profile. Papyrus provides a tool for the automatic generation of these view pages. However some of the generated pages might have to be manually edited in order to display the correct graphical element to manipulate certain properties. The Papyrus documentation provides a complete guide for the customization and the generation of property view pages[3]. Figure III-9 shows how we have filled the extension for the registration of property view pages.

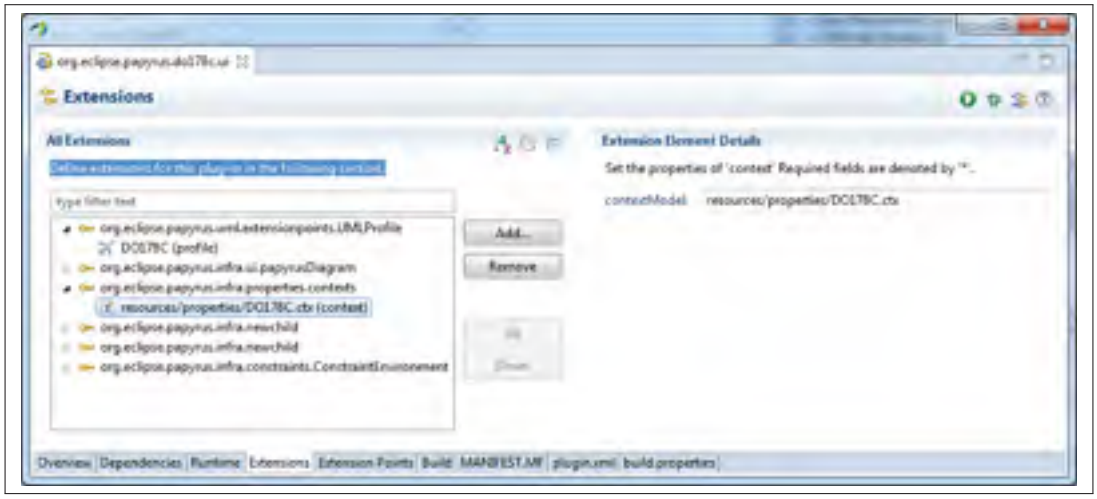


Figure-A III-9 Property view pages plug-in registration.

3. Extending Papyrus to include the profile templates

The `org.eclipse.papyrus.do178c.wizard` plug-in is in charge of the registration of template models within the papyrus environment. A template is a base model that is copied into a newly created UML model. Such templates are created by the designer of the profile. In our case, the templates contain predefined models for the definition of the software life cycle for each software level.

This plug-in uses the following extension point provided by the Papyrus environment:

[3] "Properties view customization": http://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.papyrus.views.properties.doc%2Ftarget%2Fgenerated-eclipse-help%2Fproperties-view.html

- *org.eclipse.papyrus.uml.diagram.wizards.templates*: This extension point registers UML2 model templates that can be used when creating a new model. The content of the template is then copied into the newly created model. Figure III-10 shows the customized Papyrus wizard for selecting a template. In this case, we defined four templates, one for each software level. Figure III-11 shows the extension point as filled for our needs.

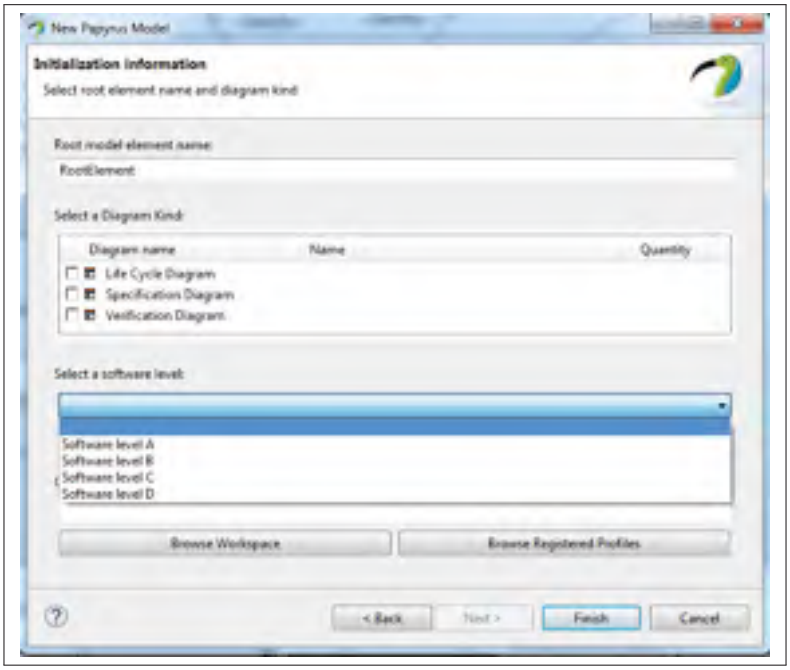


Figure-A III-10 Customized Papyrus wizard for the selection of a template.

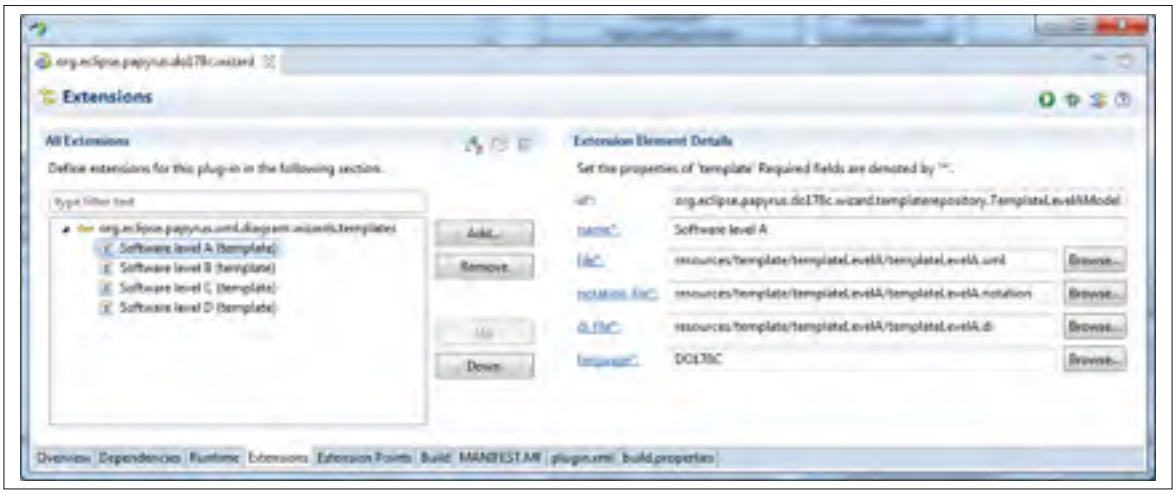


Figure-A III-11 UML model template registration.

4. Extending Papyrus to include the Profile viewpoint

Viewpoints allow the customization and specialization of the user experience by modifying the tool graphical aspects that related to new diagrams and tables. Viewpoints offer the possibility to:

- constrain the available diagrams and tables to particular users
- implement new kind of diagrams that define custom names, icons, figures, palettes, and custom display of stereotypes through CSS style sheets for domain specific views

The `org.eclipse.papyrus.do178c.viewpoint` plug-in is in charge of registering the DO178C viewpoint to which we will be able to later attach the profile's diagrams. The plug-in uses the following extension point:

- *org.eclipse.papyrus.infra.viewpoints.policy.custom*: This extension allows to register a new viewpoint configuration. The definition of a viewpoint configuration is realized either by a configuration or contribution. The differences between the two are explained in the Papyrus user guide related to viewpoint[4]. The only detail that was not explained in the aforementioned guide was that we had to load within our configuration file the following configuration file defined by the environment *platform:/plugin/org.eclipse.papyrus.infra.viewpoints.policy/-builtin/default.configuration*. This does not define new diagram kind. It only defines the new view category for DO178C to which we later attach our diagrams. Figure III-12 provides an example of how we have filled the required input for the extension point using a contribution.

[4] "Viewpoints in Papyrus": http://help.eclipse.org/neon/topic/org.eclipse.papyrus.infra.viewpoints.doc/target/generated-eclipse-help/viewpoints.html?cp=66_1_0

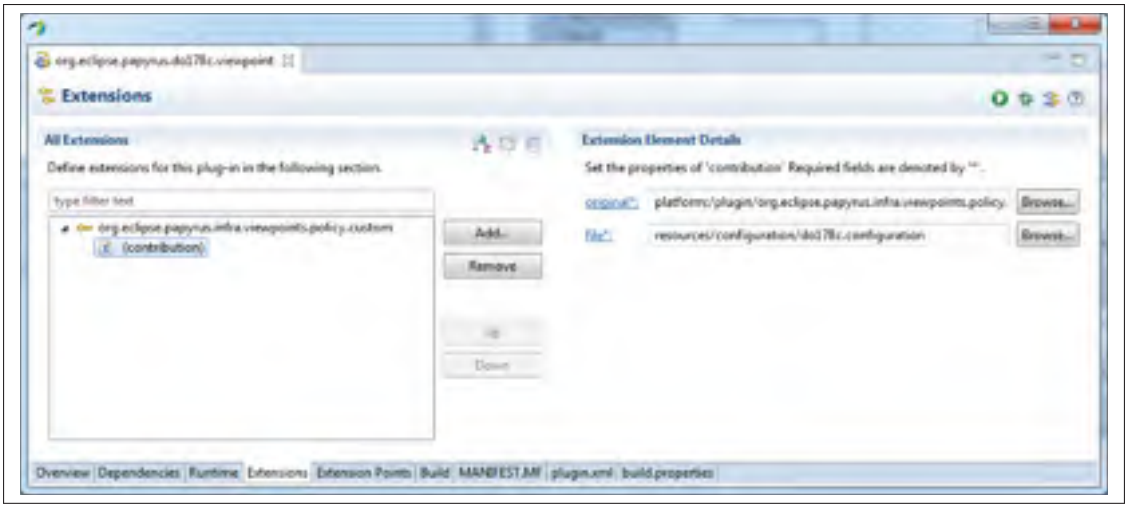


Figure-A III-12 DO178C viewpoint registration.

5. Extending Papyrus to include the profile's diagrams

These plug-ins are in charge of the definition of new diagrams for DO178C. We have created one plug-in for each newly defined diagram. The list of plug-ins is the following:

- `org.eclipse.papyrus.do178c.diagram.requirement`
- `org.eclipse.papyrus.do178c.diagram.lifecycle`
- `org.eclipse.papyrus.do178c.diagram.verification`

Each of these plug-ins are built following the same steps. These plug-ins define new diagram kinds along with their custom palettes, icons, CSS style sheet. These plug-ins are created using the instructions related to viewpoint customization[5].

- *org.eclipse.papyrus.infra.viewpoints.policy.custom*: This extension registers a diagram configuration file. A configuration file defines various properties of the diagram such as the palette definition file, the CSS style sheet for the diagram, and the elements of the UML metamodel that can be represented in the diagram. Figure III-13 shows the extension done to include the software life cycle diagram. The diagram configuration needs to access properties defined in the configuration file of the viewpoint plug-in in order to attach the diagram definition to the DO178 viewpoint.

[5] "Viewpoints in Papyrus": http://help.eclipse.org/neon/topic/org.eclipse.papyrus.infra.viewpoints.doc/target/generated-eclipse-help/viewpoints.html?cp=66_1_0

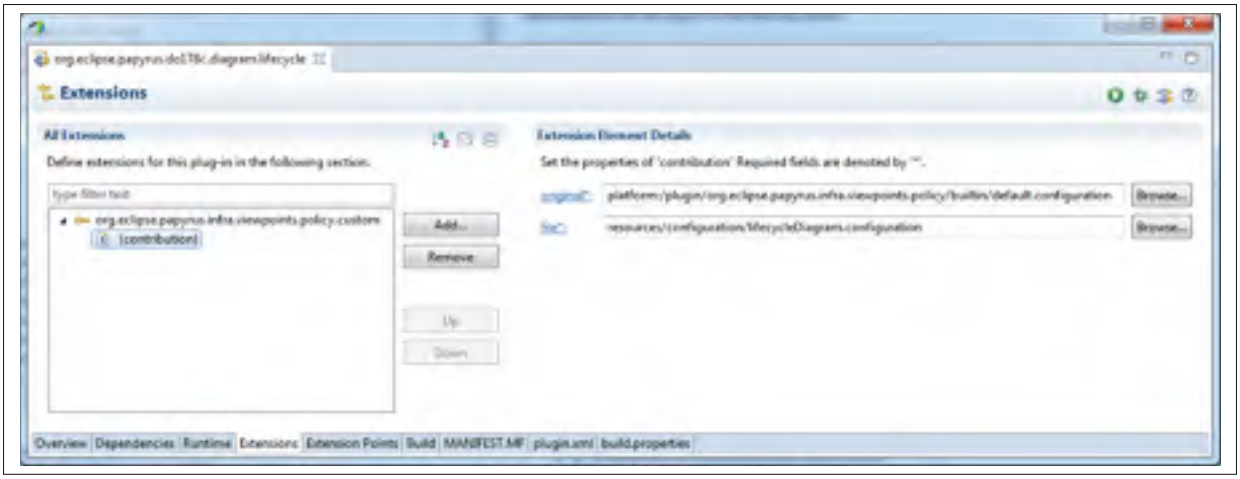


Figure-A III-13 Diagram configuration file registration.

6. Extending Papyrus to include the profile constraints

The `org.eclipse.papyrus.do178c.validation` plug-in is in charge of registering the constraints defined by our profile in order to verify the well-formedness rules of a model that uses our profile. This plug-in uses the EMF validation framework and simply registers all of our constraints that have been written using the Java programming language.

Papyrus provides a tool that help in the generation of a skeleton for this plug-in. This tool, called Papyrus DSML validation, needs to be installed first. Figure III-14 shows the Papyrus wizard to install this tool. This tool generates additional code that is used by the validation framework to identify the context of model elements parsed by the validation engine in order to assess which constraint applies to the element being verified by the engine.

Figure-A III-14

Because our constraints are not defined within the profile model, we had to add manually every information related to each constraint (the Java classes implementing the constraints, and the model element to which the constraint applies).

This plug-in uses the following extension points:

- *org.eclipse.emf.validation.constraintProviders*: This extension point enables the registration of the constraint as a Java class. An example of how the extension point was used is provided in Figure III-15. It specifies the constraint severity, its language, the class that implements the constraint, and its ID used by the tool for registration.

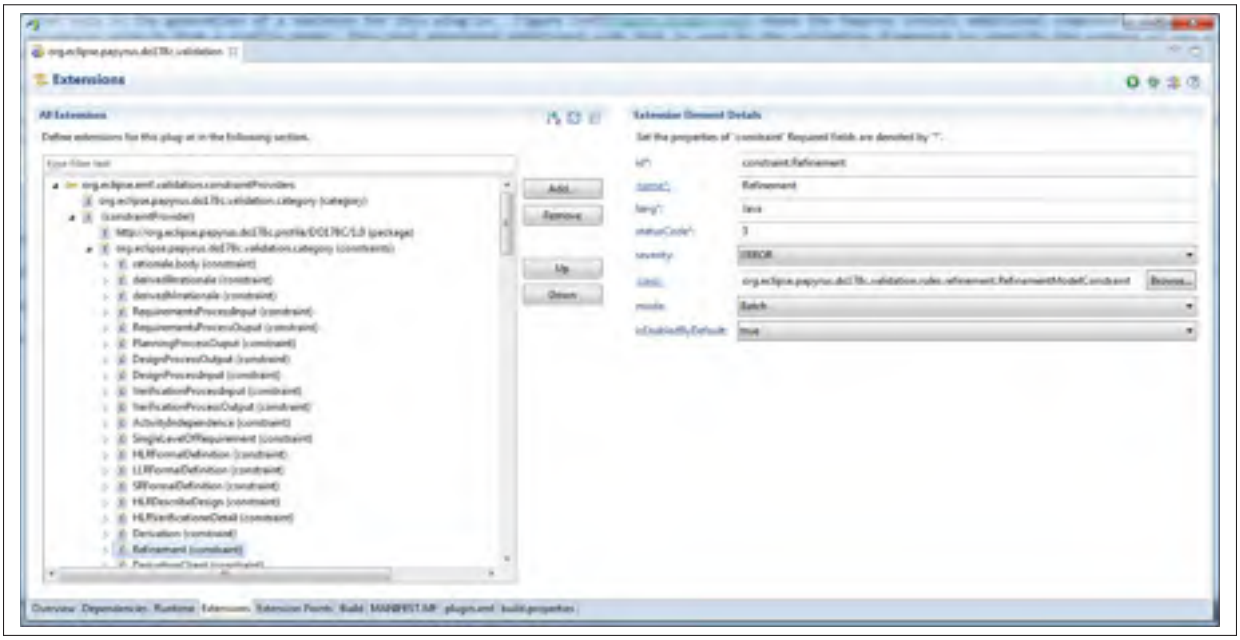


Figure-A III-15 Example of the constraintProvider definition.

- *org.eclipse.emf.validation.constraintBindings* This extension point is in charge of registering for each execution context the applicable constraints. An example of how the extension point was used is provided in Figure III-16.

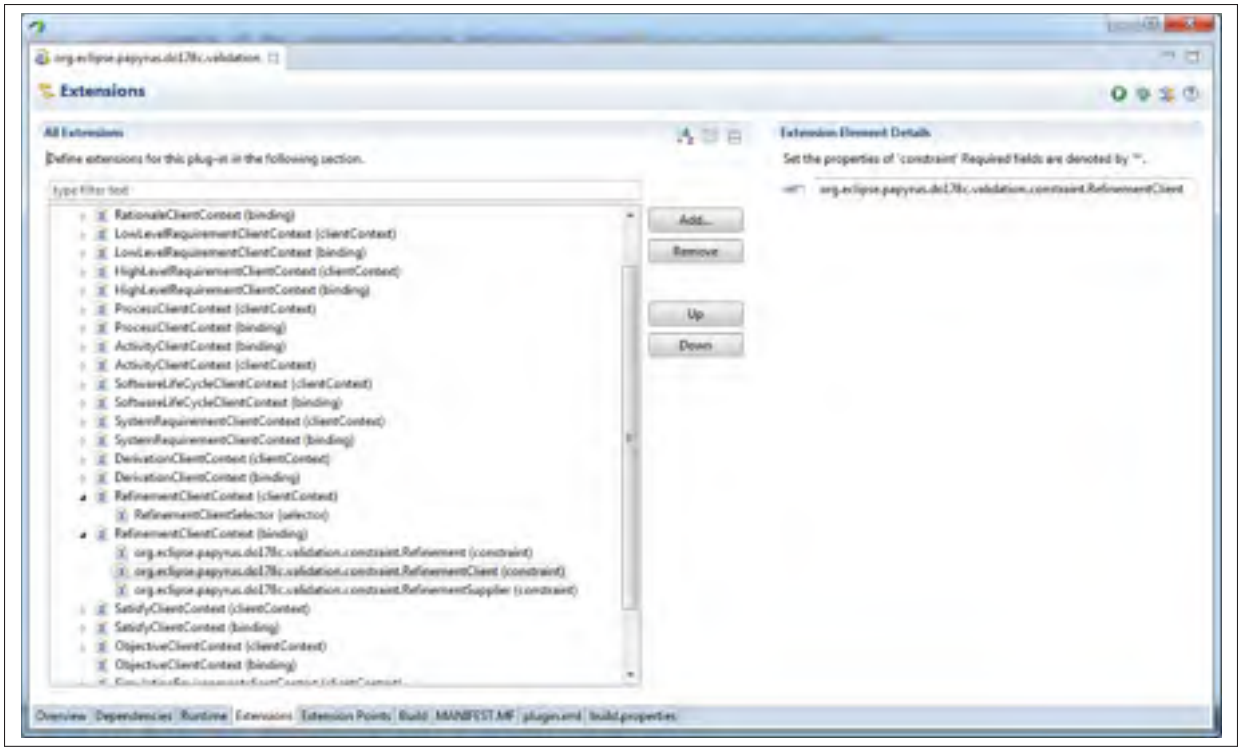


Figure-A III-16 Example of the constraintBinding definition.

7. Modifications made to Papyrus plug-ins

To customize Papyrus behavior, we modified the `org.eclipse.papyrus.uml.diagram.wizards` plug-in. We modified this wizard in order to be able to disable the selection of a diagram kind when a software life cycle template model has been selected as shown on Figure III-17, where the diagram selecting has been greyed out once a template has been selected.

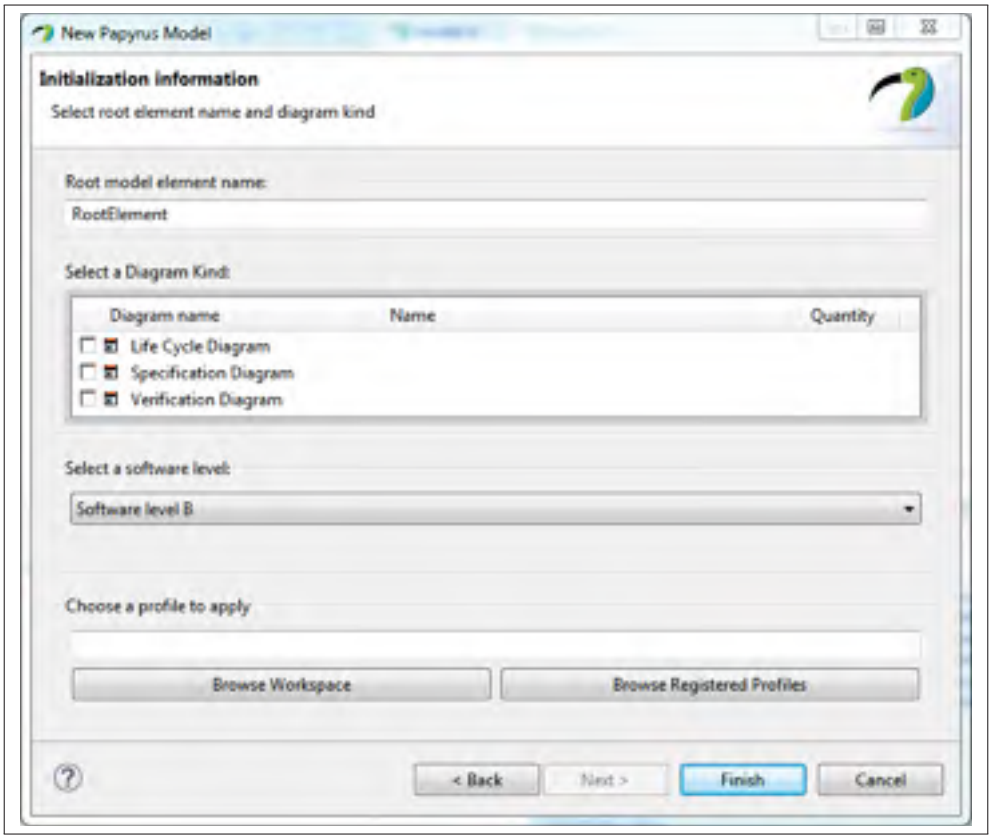


Figure-A III-17 Disabled diagram kind selection when a template is selected.

BIBLIOGRAPHY

Atkinson, C. & Kuhne, T. (2003). Model-driven development: a metamodeling foundation. *Ieee software*, 20(5), 36-41. doi: 10.1109/MS.2003.1231149.

Berkenkötter, K. & Hannemann, U. (2006). Computer safety, reliability, and security: 25th international conference, safecomp 2006, gdansk, poland, september 27-29, 2006. proceedings (pp. 398-411).

Boniol, F. & Wiels, V. (2014). The landing gear system case study. In *ABZ 2014: The Landing Gear Case Study: Case Study Track, Held at the 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z, Toulouse, France, June 2-6, 2014. Proceedings* (pp. 1–18). Springer International Publishing. doi: 10.1007/978-3-319-07512-9_1.

De la Vara, J. L., Ruiz, A., Attwood, K., Espinoza, H., Panesar-Walawege, R. K., Lopez, A., del Rio, I. & Kelly, T. (2016). Model-based specification of safety compliance needs for critical systems: A holistic generic metamodel. *Information and software technology*, 72, 16 - 30.

Djurić, D., Gašević, D. & Devedžić, V. (2005). Ontology modeling and mda. *Journal of object technology*, 4(1), 109–128.

FAA. (1988). AC 25.1309-1A System Design and Anaysis.

FAA. (2013). AC 20.115C Airborne Software Assurance.

Fuentes-Fernández, L. & Vallecillo-Moreno, A. (2004). An introduction to uml profiles. *Uml and model engineering*, 2.

Gallina, B. & Andrews, A. (2016). Deriving verification-related means of compliance for a model-based testing process. *Digital avionics systems conference (dasc), 2016 ieee/aiaa 35th*, pp. 1–6.

Heimdahl, M. P. (2007). Safety and software intensive systems: Challenges old and new. *2007 future of software engineering*, pp. 137–152.

Huhn, M. & Hungar, H. (2010). Uml for software safety and certification: Model-based development of safety-critical software-intensive systems. *Proceedings of the 2007 international dagstuhl conference on model-based engineering of embedded real-time systems*, (MBEERTS'07), 201–237.

Kuschnerus, D., Bruns, F., Bilgic, A. & Musch, T. (2012). A uml profile for the development of iec 61508 compliant embedded software. *Proceedings of the 6th international congress and exhibition—embedded real time software and systems, erts2 2012*.

Lagarde, F., Espinoza, H., Terrier, F., André, C. & Gérard, S. (2008). Leveraging patterns on domain models to improve uml profile definition. *Proceedings of the theory and practice of software, 11th international conference on fundamental approaches to software engineering*, (FASE'08/ETAPS'08), 116–130. Consulted at http://dl.acm.org/citation.cfm?id=1792838.1792851.

Langer, B. & Tautschnig, M. (2008). Navigating the requirements jungle. *International symposium on leveraging applications of formal methods, verification and validation*, pp. 354–368.

Lempia, D. L. & Miller, S. P. (2009). Requirements engineering management handbook. *National technical information service (ntis)*, 1.

Marques, J. C., Yelisetty, S. M. H., Dias, L. A. V. & da Cunha, A. M. (2012, April). Using model-based development as software low-level requirements to achieve airborne software certification. *Information technology: New generations (itng), 2012 ninth international conference on*, pp. 431-436.

Mathworks. (2017). Do178 case study. Consulted at https://www.mathworks.com/matlabcentral/fileexchange/56056-do178-case-study?focused=6804533&tab=example.

Nair, S., de la Vara, J. L., Melzi, A., Tagliaferri, G., De-La-Beaujardiere, L. & Belmonte, F. (2014). Safety evidence traceability: Problem analysis and model. *International working conference on requirements engineering: Foundation for software quality*, pp. 309–324.

Nejati, S., Sabetzadeh, M., Falessi, D., Briand, L. & Coq, T. (2012). A sysml-based approach to traceability management and design slicing in support of safety certification: Framework, tool support, and case studies. *Inf. softw. technol.*, 54(6), 569–590. doi: 10.1016/j.infsof.2012.01.005.

Object Management Group (OMG). (2011). UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems.

Object Management Group (OMG). (2014a). UML Profile for BPMN Processes, Version 1.0.

Object Management Group (OMG). (2014b). Object Constraint Language (OCL) Specification, Version 2.4.

Object Management Group (OMG). (2014c). OMG System Modeling Language (OMG SysML), Version 1.4.

Object Management Group (OMG). (2014d). UML Profile for Advanced and Integrated Telecommunication Services (TelcoML), Version 1.0.

Object Management Group (OMG). (2014e). UML Testing Profile (UTP), Version 1.2.

Object Management Group (OMG). (2015a). Meta-Object Facility (MOF) Specification, Version 2.5.

Object Management Group (OMG). (2015b). Unified Modeling Language (UML) Specification, Version 2.5.

Panesar-Walawege, R. K., Sabetzadeh, M. & Briand, L. (2013). Supporting the verification of compliance to safety standards via model-driven engineering: Approach, tool-support and empirical validation. *Inf. softw. technol.*, 55(5), 836–864.

Paz, A. & El Boussaidi, G. (2017). *Landing gear control software: An avionics software development case study*. Montreal, Canada. Consulted at http://dx.doi.org/10.13140/RG.2.2.34900.19848.

Pettit, R. G., Mezcciani, N. & Fant, J. (2014). On the needs and challenges of model-based engineering for spaceflight software systems. *Object/component/service-oriented real-time distributed computing (isorc), 2014 ieee 17th international symposium on*, pp. 25–31.

RTCA. (2011a). DO-178C Software Considerations in Airborne Systems and Equipment Certification.

RTCA. (2011b). DO-330 Software Tool Qualification Considerations. RTCA & EUROCAE.

RTCA. (2011c). DO-331 Model-Based Development and Verification Supplement to DO-178C and DO-278A. RTCA & EUROCAE.

RTCA. (2011d). DO-332 Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A. RTCA & EUROCAE.

RTCA. (2011e). DO-333 Formal Methods Supplement to DO-178C and DO-278A. RTCA & EUROCAE.

Rushby, J. (2011). New challenges in certification for aircraft software. *Proceedings of the ninth acm international conference on embedded software*, (EMSOFT '11), 211–218. doi: 10.1145/2038642.2038675.

Schmidt, D. C. (2006). Model-driven engineering. *Computer-ieee computer society-*, 39(2), 25.

Selic, B. (2007). A systematic approach to domain-specific language design using uml. *Proceedings of the 10th ieee international symposium on object and component-oriented real-time distributed computing*, (ISORC '07), 2–9. doi: 10.1109/ISORC.2007.10.

Stallbaum, H. & Rzepka, M. (2010). Toward do-178b-compliant test models. *2010 workshop on model-driven engineering, verification, and validation*, pp. 25-30.

Viatra. (2017). Viatra - scalable reactive model transformations - eclipse. Consulted at https://www.eclipse.org/viatra/.

Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L. C. L., Visser, E. & Wachsmuth, G. (2013). *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org.

Wu, J., Yue, T., Ali, S. & Zhang, H. (2015). A modeling methodology to facilitate safety-oriented architecture design of industrial avionics software. *Software: Practice and experience*, 45(7), 893–924. doi: 10.1002/spe.2281.

Zoughbi, G., Briand, L. & Labiche, Y. (2010). Modeling safety and airworthiness (rtca do-178b) information: conceptual model and uml profile. *Software & systems modeling*, 10(3), 337–367.