A DO-178C-compliant Model-Driven Approach to Support the Development and Certification of Safety-Critical Avionics Software

by

Andrés Felipe PAZ LOBOGUERRERO

THESIS PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
IN PARTIAL FULFILLMENT FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
Ph.D.

MONTREAL, FEBRUARY 14, 2020

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, Thesis Supervisor
Département de Génie logiciel et des technologies de l'information
École de Technologie Supérieure

Mr. Antoine Tahan, President of the Board of Examiners
Département de génie mécanique
École de Technologie Supérieure

Mr. Abdelouahed Gherbi, Member of the jury
Département de Génie logiciel et des technologies de l'information
École de Technologie Supérieure

Mr. Yvan Labiche, External Independent Examiner
Systems and Computer Engineering
Carleton University

THIS THESIS WAS PRESENTED AND DEFENDED

IN THE PRESENCE OF A BOARD OF EXAMINERS AND THE PUBLIC

ON FEBRUARY 4, 2020

AT ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

to the memory of my beloved grandparents, Cielo and Eduardo

to my dear parents, Cielo and Diego

PREFACE

The research presented hereinafter was conducted in the Laboratoire en Architecture de Système Informatiques at École de Technologie Supérieure, Université du Québec. To the best of my knowledge, this work is original, except where references are made to previous work. Neither this, nor any substantially similar dissertation has been or is being submitted for any other degree, diploma or other qualification at any other university.

Parts of this work have been presented in the following publications:

A. Paz, G. El Boussaidi and H. Mili, "*checsdm*: A Method for Ensuring Consistency in Heterogeneous Safety-Critical System Design," in IEEE Transactions on Software Engineering (Early Access), January 2020.

N. Metayer, A. Paz and G. El Boussaidi, "Modelling DO-178C Assurance Needs: A Design Assurance Level-Sensitive DSL," in Proceedings of the 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), October 2019.

A. Paz and G. El Boussaidi, "Supporting Consistency in the Heterogeneous Design of Safety-Critical Software," in Proceedings of the 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), July 2019, pp. 37–46.

A. Paz and G. El Boussaidi, "A Requirements Modelling Language to Facilitate Avionics Software Verification and Certification," in Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering Workshops (ICSEW), May 2019, pp. 1–8.

A. Paz and G. El Boussaidi, "Building a Software Requirements Specification and Design for an Avionics System: An Experience Report," in Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC'18), April 2018, pp. 1262–1271.

A. Paz and G. El Boussaidi, "On the Exploration of Model-Based Support for DO-178C-Compliant Avionics Software Development and Certification," in Proceedings of the 2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), October 2016, pp. 229–236.

ACKNOWLEDGEMENTS

My deepest thanks go to my supervisor, Ghizlane El Boussaidi, for an enjoyable and motivating research experience as well as for the trust she placed in me to carry out this work. I am very grateful to her for guiding, encouraging and supporting me throughout the development of this thesis with great competence and scientific rigour.

This research was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) (CRDPJ 463076-14) and the Consortium for Research and Innovation in Aerospace in Quebec (CRIAQ) under project AVIO-604. I thank again my supervisor, Ghizlane El Boussaidi, for her financial support throughout the duration of the research. I would like to thank the team of project AVIO-604, especially Pierre, Claire, Martin, Nicolas and Alexis, for their comments and feedback throughout the project, and for helping in the evaluation of the research results.

My thanks also go to Hafedh Mili, professor at Université du Québec à Montréal, who was of such a great assistance in shaping this work.

I thank the members of the board of examiners for agreeing to evaluate this thesis. I am very thankful to Yvan Labiche, professor at Carleton University, for serving as the external independent examiner and for his useful and constructive comments and feedback.

I owe a debt of gratitude to my great friend Hugo Arboleda for preparing me for and encouraging me into pursuing a Ph.D. I will always treasure the life lessons you gave me during the time we spent working together. I would like to thank my friend Camilo Valderrama for his kind words of support.

A huge thank you goes to my parents, Cielo and Diego, and my brother, Gustavo, for their unconditional love, support and encouragement. I am extremely grateful to my late grandparents, Cielo and Eduardo, who always supported my studies and gave me so much. I would have never gotten this far without any of you.

Une Approche Orientée Modèle et Conforme à la Norme DO-178C pour Appuyer le Développement et la Certification des Logiciels d'Avionique Critiques

Andrés Felipe PAZ LOBOGUERRERO

RÉSUMÉ

Les systèmes logiciels, de plus en plus complexes à l'heure actuelle, sont conçus pour répondre aux besoins des domaines dans lesquels la sûreté est cruciale, comme les aéronefs. Cependant, l'ingénierie de logiciels d'avionique n'est pas une tâche simple. D'un côté, les consommateurs demandent l'intégration des fonctionnalités, souvent complexes, pour les rendre plus intelligents. D'un autre côté, leurs environnements opérationnels ont des exigences de sûreté. Pour cette raison, leur ingénierie est hautement réglementée afin de s'assurer qu'ils seront développés d'une manière appropriée pour éviter de nuire aux personnes et aux biens dans leurs environnements opérationnels. Les fournisseurs de systèmes d'avionique doivent fournir des preuves que leurs systèmes sont conformes à la norme réglementaire applicable DO-178C. Ces deux facteurs ont mis en évidence le besoin de techniques d'ingénierie permettant de réduire la complexité du développement et d'appuyer les efforts de certification. Dans ce contexte, l'ingénierie dirigée par les modèles (IDM) représente une alternative plus économique en temps et en coûts. Ceci est principalement dû au fait que les modèles représentent l'information à des niveaux d'abstraction appropriés permettant la manipulation et l'analyse de cette information tout au long du cycle de développement logiciel. Néanmoins, il reste encore des défis importants à relever pour que l'IDM puisse soutenir de manière globale le développement et la certification des systèmes d'avionique, et aussi pour qu'elle soit l'objet d'une adoption à grande échelle.

Cette thèse se focalise sur l'investigation de l'IDM pour développer des systèmes hétérogènes où plusieurs langages de modélisation sont utilisés pour exprimer différents aspects du même système. Dans ce contexte, nous investiguons l'utilisation de l'IDM pour assurer: 1) la cohérence entre les modèles de conception, 2) la traçabilité des élements de ces modèles par rapport aux exigences, et 3) une relation explicite entre toutes ces informations et les objectifs réglementaires de la DO-178C. Pour faire face à ces défis, nous proposons une méthode, basée sur l'IDM, qui supporte la cohérence entre modèles de conception hétérogènes et fournit une infrastructure de spécification des exigences. Toutes les données produites par cette méthode sont traçables aux objectifs réglementaires de la DO-178C.

Afin d'arriver à cette contribution, nous avons examiné diverses approches de modélisation pour la conception hétérogène et la gestion de la cohérence. Nous avons construit une méthode systématique et automatisée pouvant être appliquée à une variété de scénarios de conception impliquant différents langages de modélisation et directives de conception. Les méthodes de modélisation de conception hétérogènes existantes peuvent être utilisées dans la méthode proposée. En outre, nous avons étudié certains langages de modélisation des exigences utilisés dans le contexte des systèmes critiques. Nous avons analysé le support que ces langages offrent pour faciliter la certification et capturer des informations structurées et sémantiquement

riches permettant des analyses et des tests basés sur les exigences. En se basant sur les résultats de cette étude, nous avons construit un langage de modélisation des exigences qui combine certain nombre des langages étudiés et qui s'intègre dans notre méthode. À ce propos, nous avons suivi une approche systématique pour construire un tel langage de modélisation. Nous avons réalisé des évaluations empiriques de notre méthode par le biais d'études de cas. Nous avons fait également des ateliers avec des praticiens industriels en vue d'évaluer leurs perceptions de la méthode proposée.

Les résultats obtenus dans ces évaluations nous permettent de conclure que la méthode proposée dans cette thèse, peut être utilisée efficacement pour appuyer le développement et la certification des logiciels d'avionique selon la norme DO-178C. En particulier, les praticiens industriels considèrent qu'elle est facile à comprendre et qu'il y a une forte probabilité qu'ils l'adoptent dans le contexte de leur travail.

Mots-clés: Ingénierie dirigée par les modèles, systèmes critiques, systèmes d'avionique, conception hétérogène, spécification des exigences, certification, DO-178C.

A DO-178C-compliant Model-Driven Approach to Support the Development and Certification of Safety-Critical Avionics Software

Andrés Felipe PAZ LOBOGUERRERO

ABSTRACT

Increasingly complex software is nowadays engineered to cater to safety-critical domains, like aircraft. However, engineering software for avionics systems is not a straightforward task. On one hand, consumer demands have spurred the need to pack features in, often intricate ones that will make systems smarter. On the other hand is the safety-critical nature of their operational environments. Because of this, their engineering is highly regulated in order to ensure they are developed appropriately to avoid, or at least mitigate, posing undue harm to anyone or anything in their operational environments. Avionics systems manufacturers are, thus, obliged to provide the appropriate software safety assurance in compliance with the applicable regulatory norm DO-178C. These two factors have pointed toward the need of engineering techniques that aid in reducing development complexities and support certification endeavours. Model-Driven Engineering (MDE) has been proposed as a cost- and time-effective alternative. The main rationale behind MDE is that models represent information at the right levels of abstraction to enable reasoning and ease information manipulation throughout the entire engineering life cycle. Nonetheless, significant challenges must still be overcome for MDE to comprehensively support the development and certification of avionics systems, and experience widespread adoption.

This thesis focuses on the investigation of MDE for supporting the use of different modelling languages for describing the same system while ensuring 1) consistency between the different system models, 2) traceability of the elements in such models back to requirements, and 3) all of this information explicitly relates to DO-178C regulatory objectives. Given these challenges, the main contribution of this thesis is a systematic and automated method, based on MDE, for assisting engineering teams in ensuring consistency of heterogeneous design models and providing such teams with a requirements specification infrastructure. All outputs of the method are explicitly traceable to DO-178C regulatory objectives.

To arrive at this contribution, we reviewed various heterogeneous design modelling and consistency management approaches. We derived a systematic and automated method that can be applied to various design scenarios involving different modelling languages and different design guidelines. Existing heterogeneous design modelling approaches can be leveraged within our proposed method. Furthermore, we studied several requirements modelling languages used in the context of safety-critical systems and characterized their support toward enforcing information for DO-178C certification and capturing structured semantically-rich information to enable requirements-based analyses and testing. Based on the results from such a survey we advocated for the combination of several of these languages to build a modelling language that could be integrated into the proposed method. We followed a systematic approach to build such a modelling language. We undertook empirical evaluations of our proposal by applying

it in two case studies. As part of the evaluations we also include assessment workshops with practitioners from industry to examine their perceptions about it.

The results from the empirical evaluations show that our proposal can be effectively used to support the development and certification of avionics software in accordance with DO-178C. Practitioners from industry consider the proposal to be easy to understand and gave it an overall likelihood of adoption within the contexts of their work.

Keywords: Model-Driven Engineering, safety-critical systems, avionics systems, heterogeneous design, requirements specification, certification, DO-178C.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Page

LISTINGS

Page

LIST OF ABREVIATIONS

AADL	Architecture Analysis and Design Language
API	Application programming interface
Breesse	Bridge for the Eclipse Modeling Framework ecosystem and the MathWorks Simulink and Stateflow ecosystem
CFC	Contribution to failure condition
checsdm	Consistency of Heterogeneous Embedded Control System Design Models
checsdm4a/ss	Consistency of Heterogeneous Embedded Control System Design Models for AADL, Simulink and Stateflow
checsdm4uss	Consistency of Heterogeneous Embedded Control System Design Models for UML, Simulink and Stateflow
CRIAQ	Consortium de Recherche et d'Innovation en Aérospatiale au Québec
DSML	Domain-specific modelling language
EASA	European Aviation Safety Agency
ECL	Epsilon Comparison Language
ECS	Elevator control system
EMF	Eclipse Modeling Framework
EOL	Epsilon Object Language
EUROCAE	European Organisation for Civil Aviation Equipment
EV	hydraulic electro-valve
FAA	US Federal Aviation Administration

FCS	Flight control system
GQM	Goal-question-metric
HCT	Horizontal Condition Tables
HLR	High-level requirement
IDM	Ingénierie dirigée par les modèles
JWI	Java WordNet interface
kPa	kilo-Pascal
LGCS	Landing gear control software
LGS	Landing gear system
LLR	Low-level requirement
LTL	Linear Temporal Logic
MAAB	MathWorks Automotive Advisory Board
MARTE	Modeling and Analysis of Real-Time Embedded Systems
MDE	Model-Driven Engineering
MOF	Meta-Object Facility
MPM	Multi-Paradigm Modeling
OCL	Object Constraint Language
OEM	Original Equipment Manufacturers
OMG	Object Management Group
OOT	Object-Oriented Technology

OSLC	Open Services for Lifecycle Collaboration
PBR	Property-based requirement
PID	Proportional-integral-derivative
PIM	Platform-independent model
PLE	Product Line Engineering
PSAC	Plan for Software Aspects of Certification
PSM	Platform-specific model
pw	Person week(s)
RAF	Reference Assurance Framework
RDAL	Requirements Definition and Analysis Language
RQ	Research question
RSML	Requirements State Machine Language
RTCA	Radio Technical Commission for Aeronautics
SACM	Structured Assurance Case Metamodel
SH	System hazard
SMT	Satisfiability modulo theory
SpeAR	Specification and Analysis of Requirements
SpecML	Requirements Specification Modeling Language
SRATS	System requirement allocated to software
SysML	Systems Modeling Language

TCAS II	Traffic Collision Avoidance System level II
TDL	Test Description Language
UCM	Use Case Map
UI	User Interface
UML	Unified Modeling Language
VQL	Viatra Query Language
V&V	Verification and Validation

INTRODUCTION

Over the past decades, safety-critical avionics systems have increasingly grown in size and complexity. They even have become decisive drivers for innovation in aircraft (acatech, 2011). The advent of software as the behavioural controller for such systems has, indeed, made this largely possible (Spitzer, 2007; Sztipanovits, 2007; Huhn & Hungar, 2007). However, engineering avionics software is a complex task. On the one hand, current consumer demands have spurred the need to pack features in. In recurrent instances, some intricate ones intended to make the systems smarter. On the other hand, safety is a major concern. Avionics software must operate in sensitive environments. Therefore, its engineering is highly regulated in order to ensure it is developed appropriately to avoid, or at least mitigate, undue harm to anyone or anything in their operational environments. Avionics systems manufacturers are, thus, obliged to provide the appropriate software safety assurance in compliance with the applicable regulatory norm DO-178C.

There has been increasing work around more up-to-date, effective engineering techniques and technologies to aid avionics systems manufacturers in reducing development complexities and to support them in their certification endeavors (Pettit *et al.*, 2014; McGregor *et al.*, 2017). The increased need for automation tools and interoperability have pointed practitioners from industry, and academics alike, toward Model-Driven Engineering (MDE). Selic (2003) defines a model as a reduced (*i.e.* simplified, abstract) representation of some (aspect of a) system that highlights its properties of interest from a given viewpoint. MDE was developed around the premise to turn models into first-class artifacts across the entire engineering life cycle in an attempt at delivering high quality systems in the most productive way (Schmidt, 2006). In order to keep up with such a premise, MDE makes use of domain-specific modelling languages (DSMLs) to provide users with a working environment where they can directly manipulate domain concepts (Kelly & Tolvanen, 2007). MDE's gained popularity has been due to its cost- and time-effectiveness (Huhn & Hungar, 2007). The main rationale explaining this is that

models represent information at the right levels of abstraction to enable reasoning and ease information manipulation throughout the entire engineering life cycle. Nonetheless, significant challenges must still be overcome for MDE to comprehensively support the development and certification of avionics systems, and experience widespread adoption in such a safety-critical domain.

The remainder of this chapter presents the precise context of our research, states the research problem that was addressed, sets out the objectives for this thesis, describes the methodology followed to achieve such objectives, and summarizes the contributions made. Finally, it outlines the organization of the rest of the dissertation.

Research Context

This thesis has been motivated in direct response to the development and certification needs of two Canadian companies as part of CRIAQ[1] project AVIO-604. The companies are large suppliers of safety-critical systems, many of which are avionics systems subject to certification compliance with DO-178C (Rad, 2011a), and its DO-331 (Rad, 2011b) and DO-332 (Rad, 2011c) supplements.

The Radio Technical Commission for Aeronautics (RTCA) along with its European counterpart, the European Organisation for Civil Aviation Equipment (EUROCAE), have issued the joint DO-178C/ED-12C guideline (Rad, 2011a). The guideline represents best development and safety practices that have been followed to produce safe avionics software systems (Esposito *et al.*, 2011; Areias *et al.*, 2014). Certification authorities around the globe such as the United States Federal Aviation Administration (FAA), the European Aviation Safety Agency (EASA) and Transport Canada have adopted DO-178C as the primary—but not the only—

[1] Consortium de Recherche et d'Innovation en Aérospatiale au Québec

compliance means for approving software *airworthiness*[2]. Hence, DO-178C associates an accredited certification schemata, meaning it establishes a stringent certification process seeking to achieve a high level of confidence in the airborne software's capabilities under normal and abnormal operations (Rad, 2011a). In particular, the document defines the development methodology as well as the proper qualitative and quantitative evidence necessary for assessing the fulfillment of the system's intended purpose (Esposito *et al.*, 2011; Ceccarelli & Silva, 2013).

The guideline largely suggests following a V-Model life cycle (Huhn & Hungar, 2007; Heimdahl, 2007; Ceccarelli & Silva, 2013; Areias *et al.*, 2014; Özçelik & Altilar, 2015; McGregor *et al.*, 2017). This is on the grounds that its sequential nature and strong emphasis on verification and validation activities and traceability for each phase promotes a meticulous design and implementation in order to build a safe system. Figure 0.1 illustrates the V-Model and its phases. The downward segment of the V-Model, the development flow, comprises three major phases or process groups: Requirements, Design, and Source Code and Executable Object Code (Rad, 2011a). The upward segment of the V-Model corresponds to Verification & Validation (V&V). V&V is considered transverse to the system development and must be requirements-based, meaning verification cases exist for checking and ensuring that the resulting system meets its specified requirements (both high-level and low-level). Other transverse processes (not illustrated) are Planning and Certification Liaison. This thesis focuses on a subset of these life cycle processes, namely Requirements and Design, and part of the V&V related to the previous two processes.

Roughly speaking, the prescribed V-Model flow is executed as follows. Development begins with the analysis of system requirements allocated to software (SRATS). Safety and hazard assessments are not performed as part of the scope of DO-178C, although their findings are taken

[2] Reliability and safe-to-use in flight.

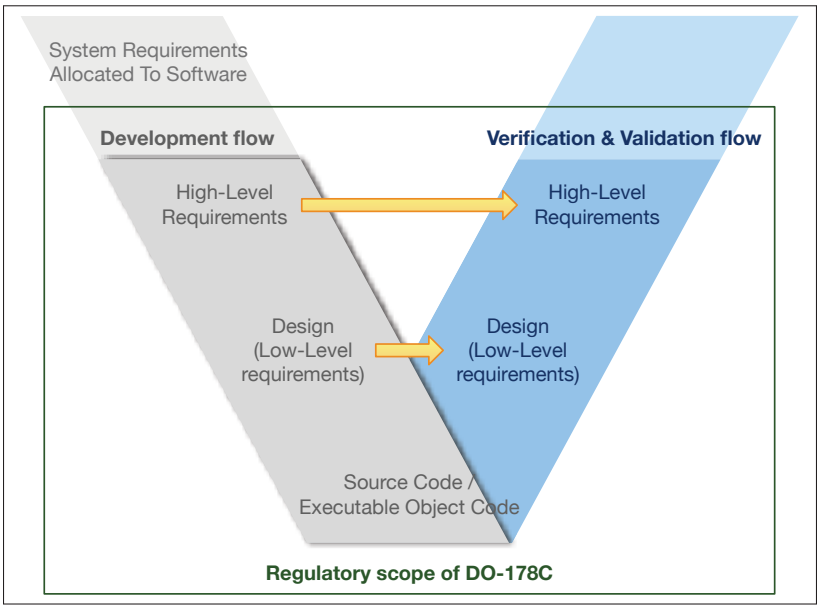


Figure 0.1 Suggested DO-178C development life cycle.

as inputs in this activity. Such SRATS must be developed (*i.e.* refined and decomposed) into high-level software requirements (HLRs). A review of the HLRs validates these against the system and safety requirements allocated to software to check if their intent is being accurately captured by the HLRs in a way suitable for directing software design activities. The correct specification of HLRs initiates the design and, in parallel, the creation of requirements-based test cases, test vectors and oracles (*i.e.* expected outputs). The design covers the definition of the architecture and the specification of low-level requirements (LLRs) (*i.e.* the detailed design). LLRs, together with the architecture, are used to guide the coding and building activities. LLRs and architecture must conform to design standards. Design standards specify methods, notations, rules, constraints, guidelines, and conventions to be used in the development of the design models. They can be specific to the company or inferred from DO-178C. Test cases are used to verify that both the LLRs and the realized software's behaviour meet the specified HLRs. In addition, reviews of the software realization itself validate that it meets the design.

One of the major changes of DO-178C from its predecessor is the included guidance on modern development and verification practices such as MDE, Object-Oriented technologies (OOT) and formal methods. All of these changes are provided as technology supplement documents, DO-331, DO-332 and DO-333, respectively, adapting the core DO-178C standard for the applicable technologies. The focus of the industry partners was on using MDE technologies to support their development and certification efforts. Thus, the DO-331 supplement became applicable. Although the DO-332 supplement is particularly intended for the software coding and integration processes, compliance with its guidance also becomes relevant when using object-oriented modelling languages like UML, which provide constructs for representing OOT features, *e.g.*, inheritance, polymorphism, overloading.

MDE has been considered for developing avionics software with different purposes in multiple ways (*e.g.*, Zoughbi *et al.* (2011); Wu *et al.* (2015); de la Vara *et al.* (2016)). Such approaches have varying aims, scopes, usages and outputs. They may focus on the following challenges in one or more of the life cycle's processes (Nair *et al.*, 2014; de la Vara *et al.*, 2016): 1) overcoming communication and understanding issues among the primary stakeholders, 2) supporting or automating development activities (*e.g.*, design), or 3) managing and collecting information for certification or compliance assessments.

Special certification considerations must be taken into account for the second challenge, *i.e.* when parts of the system under construction are to be expressed using models (Potter, 2012; Pettit *et al.*, 2014). Models may be used for software analysis, verification, simulation or code generation (Potter, 2012). Two types of models are addressed in DO-331: *specification models* and *design models*. The former represent system requirements and HLRs, and the latter represent LLRs and architecture. Engineers may leverage these different models during development, however, they must remain mindful that these models represent requirements and therefore must be treated as such under DO-178C (Potter, 2012). Models even have

to frequently coexist alongside natural language specifications. Especially, design models of avionics systems are characterized by the diversity of modelling languages used as a result of the multiple aspects that need to be captured (*e.g.*, computation, control, physical processes) (Sjöstedt *et al.*, 2008; Huang *et al.*, 2018).

Problem Statement

The research problem is framed by the conditions set by the industry partners; in essence: safety-critical avionics systems whose design is represented using different modelling languages (*e.g.*, UML, Simulink and Stateflow; AADL, Simulink and Stateflow). Thus, we state the research problem as follows:

> Effectively designing avionics systems requires the combination of diverse modelling languages for modelling the entirety of aspects that need to be covered. However, to be compliant with DO-178C, there must be explicit documentation that 1) each set of corresponding elements from different design models exhibit the same features and behaviour, 2) the design models conform to *design standards*, and 3) bidirectional traceability exists between requirements and design.

To direct our work we refined this problem into two subproblems:

P-1 Design information that is spread across multiple models and expressed in different modelling languages must be consistent.

Avionics systems are complex systems combining physical and mechanical components, networking and software (Lee, 2010). Effectively designing such systems requires a complex modelling approach that can cope with 1) dealing with diverse components, including mechanical, electronic and software, each one of these with its own underlying theories and domain vocabularies, and 2) dealing with various aspects of the same component, such as their function, structure and behaviour. It is already hard enough to relate information presented in the different software model views, *e.g.*, linking states

and transitions in a state machine to classes and methods, or system functions to packages, classes and methods. It is even harder to relate information that is spread across multiple models and expressed in different modelling languages (Lee, 2010; Eker *et al.*, 2003; Yu *et al.*, 2011; Combemale *et al.*, 2014; van den Brand & Groote, 2015). Ensuring consistency between heterogeneous design models is an important problem, in and of itself. Adding to that are the stringent quality and verification objectives and activities of certification standards, guidelines and norms. In this regard are the following four objectives: the detailed design is accurate and consistent, the detailed design conforms to design standards, the software architecture is consistent, and the software architecture conforms to design standards.

A number of existing studies have addressed mapping relationships between heterogeneous modelling languages (*e.g.*, Farkas *et al.* (2009); Sakairi *et al.* (2012); Bombino & Scandurra (2013); Ferrari *et al.* (2013); Sjöstedt *et al.* (2008); Tanaka *et al.* (2017)). However, they target pairs of specific modelling languages without devising a general approach. Other existing studies deal with consistency management in heterogeneous design models (*e.g.*, Finkelstein *et al.* (1994); Almeida da Silva *et al.* (2010); Dijkman *et al.* (2008); El Hamlaoui *et al.* (2018)) but do not explicitly provide the syntactical and semantical relationships between them. Furthermore, none of these studies tackle the issue of verifying conformance to design standards.

Establishing model consistency and adherence to design standards are resource-consuming and error-prone activities. While automated design verification and validation tools can help, they cannot be used in isolation. Indeed, if the output of such tools is not manually verified by a human being, then the tools themselves need to be *qualified*, *i.e.* they need to be subjected to the same—if not a higher—level of scrutiny as the systems they are meant to verify (Varró, 2016).

P-2 Requirements must be modelled in such a way that they exhibit an explicit relationship with the objectives and activities of DO-178C and its DO-331 and DO-332 supplements. Furthermore, modelled requirements must support the extraction of information from which reusable verification cases can be generated.

In all of the development life cycle presented in the previous section, requirements and requirements-based verification are two of the foremost development concerns for avionics software certification under DO-178C. Indeed, the most stringent compliance needs of DO-178C are centred around 1) requirements specification, 2) the proper argumentation and traceability of such requirements to design decisions and certification objectives and activities, and 3) the verification of the resulting system against the specified requirements. The majority of errors found in avionics systems usually have their origin in the Requirements phase (Feiler, 2010). Nevertheless, over 50% of them are only detected and corrected in the final phases of development (Feiler, 2010). Verification activities may end up driving a cost equivalent to seven times that of the other development activities (Moy *et al.*, 2013).

Current industry practices have requirements specified using constrained natural language and managed with the help of textual requirements databases, like IBM DOORS (Potter, 2012; Blouin, 2013). Although the use of a constrained natural language brings some discipline and rigour to requirement specifications, this practice is still focused on writing requirements in human-readable prose. This inevitably raises problems for satisfying the objectives and activities of DO-178C. Natural language indeed facilitates communication between stakeholders but it is not a suitable form of specification for supporting interrelationships, decomposition, requirements-based analyses and testing (Moy *et al.*, 2013).

Several requirements specification languages were proposed for safety-critical systems development (*e.g.*, Zoughbi *et al.* (2011); Stallbaum & Rzepka (2010); Leveson *et al.*

(1994); Micouin (2008); Blouin (2013); Bialy *et al.* (2015); Fifarek *et al.* (2017)). Others, like SysML (OMG, 2017a) and MARTE (OMG, 2011) were introduced for system and real-time system engineering, respectively, but have been employed in safety-critical systems development. Some of these languages only support the specification of natural language-based requirements (*e.g.*, OMG (2017a); Zoughbi *et al.* (2011); Stallbaum & Rzepka (2010)). Others allow the expression of semantically richer requirement statements using formal notations (*e.g.*, Leveson *et al.* (1994); Micouin (2008); Bialy *et al.* (2015); Fifarek *et al.* (2017)). However, the use of formal notations alone restrains their adoption even when they can enable requirements-based analyses and testing. Moreover, none of these languages provides sufficient support for achieving DO-178C objectives and activities, for instance, by identifying a requirement's place in DO-178C's requirements hierarchy (*i.e.* SRATS, HLR, LLR).

Research Objectives

Our main goal is to build an approach to support the development and certification of safety-critical avionics systems. In particular, we want to build on the progress made in MDE technologies to safeguard consistency of heterogeneous design models and enable requirements-based analyses and verification of safety-critical avionics systems in a way that yields evidence for certification. To do so, we refine this main objective as follows:

O-1 **Define a systematic and automated method for assisting engineering teams in ensuring consistency of heterogeneous design models of safety-critical avionics systems.**

Effective safety-critical avionics system design requires a heterogeneity of modelling mechanisms focused around specific system aspects (*e.g.*, mechanical, electronic, software), each having its own underlying domain theories and vocabularies, as well as with various aspects of the same component (*e.g.*, function, structure, behaviour). However,

the regulated nature of the avionics domain prescribes that all these design models must be consistent and conformant to design standards. We are concerned with verifying consistency of heterogeneous design models of the system under development and support evidence-gathering efforts for certification. In this regard, the requirements of the proposed systematic and automated method are the following:

a. The proposed solution shall analyse different design models and determine for each modelled element appearing in more than one of such design models if the element exhibits the same properties and behaviour.

b. The proposed solution shall analyse different design models for conformance to *design standards*.

O-2 **Develop a requirements modelling language that provides a requirements specification infrastructure for safety-critical avionics development and certification.**

Requirements engineering is a critical phase in avionics development and a prominent concern for their certification. Thus, we argue that a blended approach driven by DO-178C could be a suitable solution for expressing requirements while being easy to adopt by industry. The requirements for such an approach are the following: a. The approach shall enforce required information (*e.g.*, trace data, decomposition of requirements) for achieving objectives and activities defined in DO-178C and the DO-331 and DO-332 supplements. b. The approach shall provide facilities to capture requirements in a structured semantically-rich formalism to enable requirements-based analyses and testing. c. The approach shall deliver features to smooth the way for its adoption in industry.

O-3 **Assess the feasibility and effectiveness of our proposal when used in avionics systems.**

Our intention in this thesis is not a complete, independent path to a technological development since the requirements and costs of qualifying it for use by industry cannot be

met in academia. Instead, we deliberately focus on small elements of the engineering life cycle that can seamlessly integrate with existing technologies for filling their gaps as part of a larger toolchain. Such a result is more welcomed by the industry and seen to create the more impact to their engineering teams. Hence, we deem it important to analyze our proposed approach in terms of its feasibility and effectiveness over conventional industry practices in plausible case studies. Also, to examine the perceptions of practitioners about our approach and the likelihood they will give it for adoption in industry.

Avionics systems development and certification is an active research field. However, the legal and safety implications that public scrutiny may bring onto industry manufacturers make them keen on keeping their projects confidential. This situation hinders research and education on the engineering and certification of this type of systems. It is, therefore, necessary to have detailed and open documentation of sample systems that is readily available for research and education. It is part of this objective to elaborate such a detailed requirements specification and design for a sample avionics system.

Research Methodology

In order to achieve the objectives set out for this thesis we have used a mixed-methods approach (Creswell, 2008) combining literature reviews and case studies (Easterbrook *et al.*, 2008; Yin, 2008) over theory-develop-experiment incremental cycles. Regarding the case studies, we have followed the guidelines for conducting and reporting case study research defined by Runeson & Höst (2009).

Our research methodology comprises four phases illustrated in Figure 0.2. The first phase of the methodology consisted in performing a literature review in the interest of 1) better understanding the context of safety-critical avionics development and certification, 2) getting a hold of the extent of work done around the use of MDE in such a field, as well as in other close-

ly-related domains (*e.g.*, rail), and 3) identifying relevant existing approaches related to the scope of this thesis and sample systems that could be used for experimentation. We reviewed the current industry practices from the two partner companies in the AVIO-604 project to get an understanding of what techniques and proposals could deliver improved experiences in the work of safety-critical avionics systems engineering teams. We performed a detailed analysis of the DO-178C guideline for avionics software development to acquire a deep awareness of all the elements involved during the certification process. Later we carried out a broad exploratory study on existing model-based approaches supporting safety-critical system development and certification. In particular, we investigated a select subset of approaches having varying aims, scopes, usages and outputs to gain insight into their intents and capabilities.

The findings from this broad literature review contributed to the comprehension of safety-critical avionics software development and certification, and the identification of research gaps towards building a comprehensive MDE support for such purposes. Recall we focus on a subset of the life cycle processes, namely Requirements and Design as well as the part of the V&V process related to the verification of outputs from these processes. Thus, we then reviewed various heterogeneous design modelling and consistency management approaches. We also studied several requirements modelling languages used in the context of safety-critical systems and characterized their facilities toward enforcing required information (*e.g.*, trace data, decomposition of requirements) for achieving objectives and activities defined in DO-178C, and capturing structured semantically-rich information to enable requirements-based analyses and testing.

The second phase was derived from coming to the conclusion during the literature review phase that due to legal and safety implications that may come upon public scrutiny of a safety-critical system, the industry is keen on keeping their projects confidential. Therefore, detailed and open documentation of such systems is not readily available for research and education. This phase

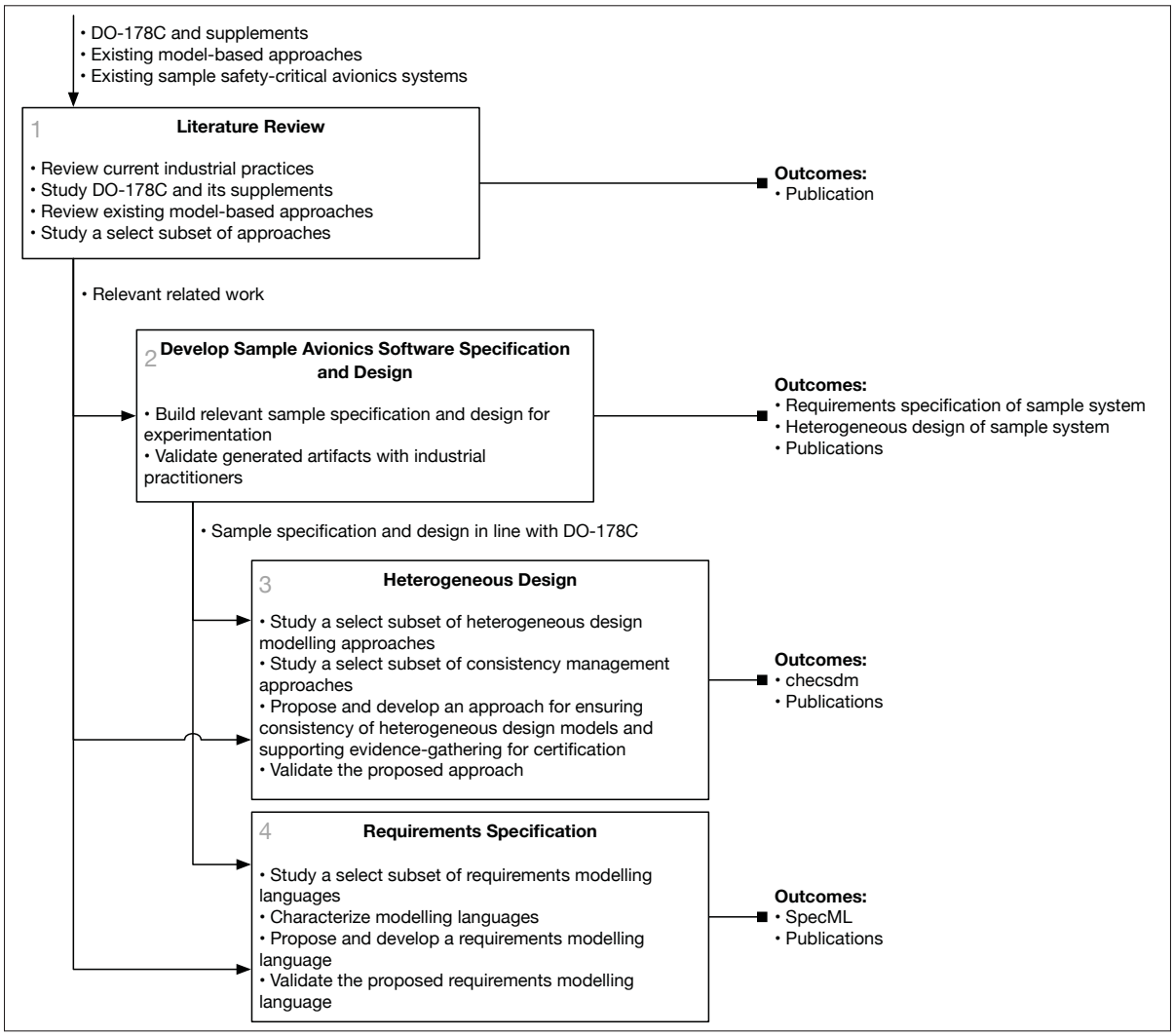


Figure 0.2 Research methodology.

was tasked with the development of a sample system specification and design for an aircraft's landing gear system according with DO-178C. The development was made in an iterative and incremental way, following an analyze-develop-validate loop. First was the definition of the system's scope and requirements, then the development of the required artifacts, and finally the validation of correctness and completeness of the generated artifacts. Close consultation with practitioners from industry was kept throughout the process to ensure the creation of a complex

and representative system. It took three iterations of SRATS and HLRs development to obtain a favorable perception about them from the practitioners involved.

The third phase is concerned with heterogeneous design modelling and consistency management. Our findings from the literature review phase revealed that there is a lack of systematic and automated methods for ensuring consistency of heterogeneous design models of safety-critical systems and supporting evidence-gathering efforts for certification. Hence, we focused on that particular task. The fourth phase centred its activities on the development of the proposed requirements modelling language. Our analysis over the characterization of existing requirements modelling languages allowed the identification of relevant features and limitations of these languages and led us to reuse as much as possible the most suitable ones. This strategy follows the Multi-Paradigm Modeling (MPM) approach (Vangheluwe *et al.*, 2002). MPM advocates for the combination, coupling and integration of independent, heterogeneous models. An advantage of MPM is the management of the complexity involved since the models are built using suitable formalisms and focused around the needed constructs for representing the targeted aspects.

During both the third and fourth phases, we undertook empirical evaluations by applying the outcomes from these phases in case studies, which involved the landing gear system developed as part of the second phase. In addition, we also conducted an assessment workshop with practitioners from industry to examine their perceptions about the outcomes of phases three and four.

Proposed Approach – in a nutshell

We propose a model-based approach to support the development and certification of safety-critical avionics systems. The approach 1) defines *checsdm*[3], a systematic and automated

[3] Pronounced "checks them".

method for assisting engineering teams in ensuring consistency of heterogeneous design models, and 2) provides SpecML, a modelling language that features a requirements specification infrastructure for DO-178C- and DO-331-compliant requirements modelling. Figure 0.3 illustrates the general flow of the approach. The approach comprises an iterative three-phased process: *elicitation*, *codification* and *operation*.

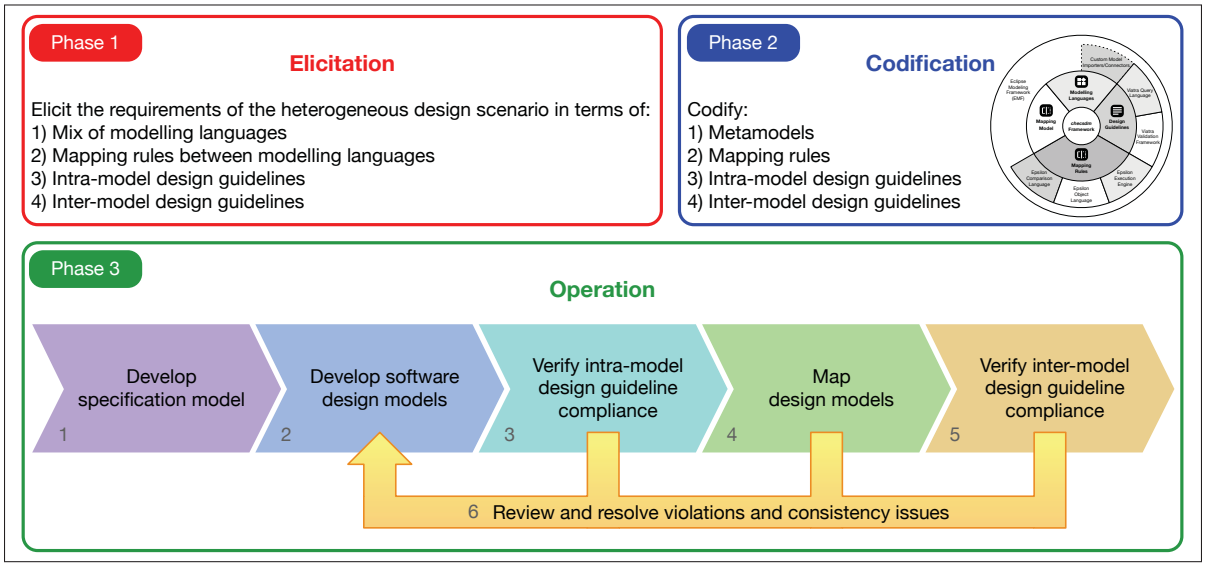


Figure 0.3 Overview of the approach.

The approach starts before any development begins, with an *elicitation* of the heterogeneous design scenario at hand in terms of 1) the mix(es) of modelling languages that are going to be used and how to use them depending on the systems' nature and the languages' purposes, 2) mapping rules between the different modelling languages, 3) intra-model design guidelines, *i.e.* design guidelines specific to models in each language taken separately, and 4) inter-model design guidelines, *i.e.* design guidelines that concern cross-model constructs. Afterwards, in the *codification* phase, engineers codify using the proposed tool framework, as required, the metamodels of the various modelling languages used, the mapping rules between the modelling languages, and the (intra- and inter-model) design guidelines. This will derive a toolchain

that will assist engineering teams in ensuring consistency of the heterogeneous design models during the *operation* phase.

The *operation* phase covers the requirements and design processes as well as the part of the verification process related to the verification of outputs from the design process. During the requirements specification process, engineers create *specification models* using SpecML to capture the system requirements. Those system requirements that are allocated to software (SRATS) are then developed (*i.e.* refined and decomposed) into high-level requirements (HLRs). In turn, HLRs can be developed into low-level requirements (LLRs) or detailed design. SpecML offers a vocabulary of constructs that is familiar to DO-178C certification. It also provides different mechanisms to capture the requirements. We introduce the use of property-based requirement (PBR) statements, which capture requirements in a structured semantically-rich formalism to enable requirements-based analyses and testing. A PBR is defined as a constraint for the system enforcing a property whenever a condition is met: *"[when condition C is met,] the value(s) of property P of object O shall be in the subset D of the set of possible values for P"*. The presence of a condition *C* is optional as indicated by the presence of square brackets. PBRs can be bound in time or to triggering events by means of timed domain constructs provided by SpecML. *Design models*, which are probably the most used mechanism of expressing LLRs/detailed design can be leveraged within SpecML through specialized traceability constructs.

During the design process, engineers create *design models* from a given operational context and a set of HLRs following the mix of modelling languages and design guidelines established during elicitation. The resulting design models are individually verified for intra-model guideline compliance. Next, correspondences between the heterogeneous design models are identified and mappings are established between overlapping elements. The analysis results are stored in a mapping model. The mapping model not only captures the relationships be-

tween overlapping elements but also flags consistency issues that were identified. Using this mapping model, the design models are verified together for inter-model guideline compliance. Following that, flagged guideline violations and consistency issues are examined and handled accordingly. This is an activity of the software design process and is intrinsically manual. The activities of the *operation* phase can be applied iteratively, until the transition criteria from the requirements and design processes to the subsequent development activities (*e.g.*, source code) have been met. Furthermore, the resulting mapping model can be used along the design models to support the subsequent development activities.

Contributions

This section summarizes the contributions of this thesis. In addition, it gives references to the publications derived from these results.

C-1 **Contributions from the literature review.**

Modelling of DO-178C. Based on the review of DO-178C and its supplements (DO-331, DO-332 and DO-333), we built a metamodel to represent its major elements and their interactions as a domain specific modelling language (DSML). The DSML is implemented as a UML profile integrating a number of constraints that enforce required information for achieving objectives and activities defined in DO-178C and its supplements. These constraints are grouped according with the possible design assurance levels relevant for certification (*i.e.* levels A through D). This work was developed with the help of a masters student and the results are detailed in his masters thesis (Metayer, 2018). The modelling of DO-178C was also presented at the 9th International IEEE Workshop on Software Certification (WoSoCer 2019) hosted at the 30th International Symposium on Software Reliability Engineering (ISSRE 2019) (Metayer *et al.*, 2019).

A characterization of model-based support for DO-178C-compliant avionics software development and certification. We present a review of a set of model-based approaches to assess their support for software development and certification under the DO-178C guideline. We built a framework to characterize these approaches according with several criteria, specially coverage of DO-178C 's required information for compliance. We analyzed the approaches using this framework and highlighted their commonalities, differences, strengths and weaknesses. Additionally, we identified open issues on which research may focus. This work has been presented at the 6th International IEEE Workshop on Software Certification (WoSoCer 2016) hosted at the 27th International Symposium on Software Reliability Engineering (ISSRE 2016) (Paz & El Boussaidi, 2016).

C-2 **Landing Gear Control Software requirements specification and design.**

We built on the landing gear system's (LGS) descriptions given by Boniol & Wiels (2014) to present the landing gear control software's (LGCS) requirements specification and design in compliance with the DO-178C guideline and the DO-331 and DO-332 supplements. We have made efforts to address inconsistencies, ambiguities and confusing wording found in the work of Boniol & Wiels (2014). We also followed the set of recommended practices on requirements engineering and management from Lempia & Miller (2009) and the work from Blouin (2013). The LGCS artifacts are accompanied by the set of methodological insights that guided the requirements specification and design. This work has been presented at the 33rd ACM Symposium on Applied Computing (SAC 2018) (Paz & El Boussaidi, 2018). The complete artifacts for the LGCS have been made available online in a technical report (Paz & El Boussaidi, 2017).

C-3 ***checsdm*: Consistency of Heterogeneous Embedded Control System Design Models.**

We defined *checsdm*, a systematic approach, based on MDE, for assisting engineering teams in ensuring consistency of heterogeneous design of safety-critical systems and

gathering evidence that will show achievement of DO-178C objectives and activities. The approach is developed as a generic *methodology* and a *tool framework*, that can be applied to various design scenarios involving different modelling languages and different design guidelines. The methodology comprises an iterative three-phased process. The first phase, *elicitation*, aims at specifying requirements of the heterogeneous design scenario. Using the proposed tool framework, the second phase, *codification*, consists in building a particular tool set that supports the heterogeneous design scenario and helps engineers in flagging consistency errors for review and eventual correction. The third phase, *operation*, applies the tool set to actual system designs. The resulting mapping model and the successful validation of design guideline compliance help support evidence-gathering efforts for showing achievement of DO-178C objectives and activities.

As sub-contributions, we present two executions of the *checsdm* approach for the specific cases of a design scenario involving a mix of UML, Simulink and Stateflow (labeled *checsdm4uss*), and a design scenario involving a mix of AADL, Simulink and Stateflow (labeled *checsdm4a/ss*). *checsdm4uss* has been presented at the 43rd IEEE Annual Computer Software and Applications Conference (COMPSAC 2019) (Paz & El Boussaidi, 2019c). The *checsdm* framework is open source (Paz & El Boussaidi, 2019f). The *checsdm* approach has been submitted and accepted with revisions as a manuscript for the journal IEEE Transactions on Software Engineering (IEEE TSE) (Paz *et al.*, 2020).

Originating from *checsdm4uss* and *checsdm4a/ss*, is another sub-contribution: *Breesse*. Both the Eclipse platform and MathWorks have successfully provided entire ecosystems and tooling for MDE. Leveraging these two MDE ecosystems for safety-critical system development would be expected. Nonetheless, these two ecosystems rarely interact due to MathWorks' closed nature and proprietary file formats. *Breesse* delivers a bridge for the Eclipse Modeling Framework ecosystem and the MathWorks Simulink and Stateflow

ecosystem. *Breesse* is open source (Paz & El Boussaidi, 2019a) and has been detailed in an unpublished manuscript (Paz & El Boussaidi, 2019d) (see Appendix V for an excerpt).

C-4 **SpecML: Requirements Specification Modelling Language.**

We developed SpecML, a hybrid modelling language extending and combining features, on the one hand, from SysML and the UML profile for MARTE, and, on the other hand, from PBR theory. Concretely, SpecML 1) enforces required information (*e.g.*, trace data, decomposition of requirements) for achieving objectives and activities defined in DO-178C, 2) captures requirements in natural language to smooth the way for its adoption in industry, and 3) provides facilities to capture requirements in a structured, semantically-rich formalism to enable requirements-based analyses and testing. SpecML is open source (Paz & El Boussaidi, 2019e). This work has been presented at the 6th International Workshop on Requirements Engineering and Testing (RET 2019) hosted at the 41st ACM/IEEE International Conference on Software Engineering (ICSE 2019) (Paz & El Boussaidi, 2019b).

Thesis Organization

This thesis is organized in six parts: Introduction, Background and Literature Review (Chapter 1), Motivating Example (Chapter 2), Proposal (Chapters 3 and 4), Evaluation (Chapter 5) and Conclusion. Following this Introduction, the Background and Literature Review chapter expands the context in which this thesis is framed and provides the state of the art. The next chapter describes a motivating example of an avionics system that will be used as a running example throughout the rest of the dissertation to illustrate key elements of our proposal. Afterwards, the two following chapters present and discuss the proposal itself. Then, the Evaluation chapter discusses the feasibility and effectiveness of our proposal when used on avionics sys-

tems. The final chapter draws conclusions and perspectives on this work. The main chapters of this thesis are described below.

Chapter 1: Background and Literature Review. The problems addressed in this thesis intersect safety-critical system certification and MDE. Regarding safety-critical system certification, this chapter introduces the DO-178C guideline on software considerations in airborne systems and equipment certification since it is a focal element of our work. Regarding MDE, this chapter introduces some definitions and key concepts. It also reviews several modelling approaches for different safety-critical system engineering activities (*e.g.*, requirements, design, certification compliance).

Chapter 2: The Landing Gear Control Software. There is a lack of reusable and comprehensive safety-critical avionics systems references in the literature that can act as benchmarks for proposals in this research field. This chapter presents a development case study of an aircraft's landing gear control software (LGCS) we undertook as part of our research. Two stages of its development are discussed: the requirement specification and the system's design. All documentation has been developed to conform with the DO-178C guideline presented in the previous chapter. The observations, challenges and issues experienced throughout the process are the most interesting to highlight from this work. Thus, these topics are discussed to motivate the need for our proposed approach.

Chapter 3: *checsdm*: Consistency of Heterogeneous Embedded Control System Design Models. This chapter is the first of two chapters dedicated to present our proposed approach. It starts by presenting *checsdm*, a generic *methodology* and a *tool framework* for verifying consistency of heterogeneous embedded control system design models. This chapter discusses these two elements in detail. It then illustrates them through a specific execution for one design scenario of our industry partners, in essence: avionics systems represented using a mix of UML, Simulink and Stateflow design models (referred to as *checsdm4uss*).

Chapter 4: SpecML: Requirements Specification Modelling Language. This chapter dives into the first step of *checsdm*'s *operation* phase, which addresses requirements specification. In this regard this chapter presents SpecML, a requirements specification modelling language providing a DO-178C-compliant documentation infrastructure. The chapter starts by showing an overview of the methodology we followed to build SpecML. It then presents the domain metamodel that stands behind SpecML to support its features. Afterwards, the chapter discusses how the concepts in this domain metamodel were mapped to the UML metamodel to build SpecML as a UML profile. The chapter ends with a look at SpecML's reference implementation.

Chapter 5: Evaluation This chapter reports on *checsdm*'s evaluation regarding its feasibility and benefits. The chapter revisits *checsdm4uss* and presents an additional execution for another design scenario involving AADL, Simulink and Stateflow design models (*checsdm4a/ss*). With respect to SpecML, the chapter provides an empirical evaluation through two avionics systems for which their requirements were modelled. One of these systems is the LGCS presented in Chapter 2. The chapter also includes the results from an assessment workshop with practitioners from industry examining their perceptions on our proposed approach.

CHAPTER 1

BACKGROUND AND LITERATURE REVIEW

The problems addressed in this thesis intersect safety-critical avionics system certification and Model-Driven Engineering (MDE). Hence, this chapter presents fundamental concepts of each of these fields as related to the scope of this dissertation. Section 1.1 gives an overview of avionics software certification under DO-178C and its supplements. Section 1.2 presents the foundational concepts of the MDE paradigm. Section 1.3 analyzes different ways of how MDE has been applied to support safety-critical software development and certification. Finally, Section 1.4 concludes this chapter with a discussion on the findings of the previous section and considerations for reusing these existing approaches or borrowing their relevant features.

1.1 Software Considerations in Airborne Systems and Equipment Certification

Avionics software can be designated as being safety-critical if its failure has a negative effect on life, property or the environment (Bozzano & Villafiorita, 2010). In order to safeguard the systems' operational environments, manufacturers must adopt a proper engineering process that can achieve and preserve safety. The DO-178C guideline has been proposed to standardize the proper qualitative and quantitative evidence necessary for assessing the fulfillment of the avionics software's intended purpose (Esposito *et al.*, 2011; Ceccarelli & Silva, 2013). DO-178C is associated with an accredited certification schemata, meaning it establishes a stringent certification process conducted by trusted third parties acting as licensing or regulatory bodies. Such a certification process seeks to achieve a high level of confidence in the avionics software's capabilities under normal and abnormal operations (Esposito *et al.*, 2011). Manufacturers develop this confidence by satisfying a series of objectives and activities that avoid, or at least mitigate, the occurrences of the system's potential contributions to failure conditions (Nair *et al.*, 2014). In the following subsections we dive into DO-178C to provide a substantial look into the compliance needs that will be the focus of this thesis.

1.1.1 DO-178C

The aim of DO-178C (Rad, 2011a) is to produce software that is validated and verified for its airworthiness, *i.e.* reliability and safe-to-use in flight. DO-178C is a conceptual guideline identifying the set of *best practices* to take into consideration during the development of software for airborne systems and equipment. These *best practices* are stated in the form of *objectives*, which have to be achieved by carrying out a set of explicitly defined *activities* that will output the acceptable evidence (*e.g.*, plans, requirements, design description), known as *data items*. The amount of objectives for which compliance must be demonstrated depends on the software's design assurance level (*software level* for short). The software level describes the severity of the system's failure conditions to which the software may contribute. DO-178C defines five software levels labeled *A* through *E*, with level *A* being the most rigorous as it requires all the objectives to be achieved and level *E* the least rigorous as it requires no objectives.

- Level *A* is assigned to catastrophic effects, meaning a failure may cause multiple fatalities and even the loss of the aircraft.
- Level *B* is assigned to hazardous/severe-major effects, meaning a failure will have a large negative effect on safety or performance causing harm to the occupants or reducing the crew's ability to operate the aircraft.
- Level *C* is assigned to major effects, meaning a failure will have a significant negative effect on safety causing inconveniences to the occupants and an increase in the crew's workload.
- Level *D* is assigned to minor effects, meaning a failure will have a slightly negative effect on safety causing some inconvenience to the occupants and an increase in the crew's workload.
- Level *E* is assigned when a failure will have no effect on safety.

DO-178C prescribes a software life cycle comprised of the following three process groups: 1) software planning process, 2) software development processes that include software requirements, software design, software coding and software integration, and 3) transverse processes

that include software verification and validation (V&V), software configuration management, quality assurance and certification liaison. Each of these processes is responsible for producing one or more *data items*.

Data items required by DO-178C include plans, standards, requirements specifications, design descriptions, and verification and trace data. One of the primary data items is the Plan for Software Aspects of Certification (commonly referred to as the PSAC). The PSAC is developed in the planning process, prior to the start of the software development processes. It is an important data item since certification bodies use it at the end of the life cycle to verify that the developed software and the process followed to develop the software, fulfilled the PSAC. Once agreement between the certification body and the applicant has been obtained on the PSAC the software's development is kick-started. The Software Development Plan is another data item specifying all the details regarding the software development process, including, for instance, development standards, environments (*i.e.* programming languages, compilers, debugging tools, and hardware used to develop and execute the software), and procedures. Software Standards (*e.g.*, requirements standards, design standards, code standards) include the definitions of methods, notations, rules, constraints, conventions to be used to develop the software requirements specifications, software architecture and code.

The Software Requirements Data data item corresponds to the specification of high-level requirements (HLRs). Likewise, the Design Description data item communicates low-level requirements (LLRs) and the software architecture. Software Verification Cases and Procedures data items contain detailed descriptions of the verification activities (*e.g.*, reviews, test case execution) that will be carried out as well as the definition of the artifacts that will be used (*e.g.*, test cases with their set of inputs, evaluated conditions, expected outputs and pass/fail criteria). Traces (*i.e.* bidirectional associations) between the contents of data items are collected as part of Trace Data data items.

Figure 1.1 presents a graphical view for the reference workflow of the processes in the software development process group and the transverse software verification and validation process. The

requirements process develops (*i.e.* refines and decomposes) system requirements allocated to software (SRATS) into HLRs suitable for directing the software design activities. The design process covers the development of LLRs from the HLRs, and the development of the software architecture. HLRs and LLRs must be traced to their originating requirement. In such cases when a requirement cannot be directly traced to an originating requirement (*e.g.*, because it specifies behaviour beyond that specified by the higher-level requirements) it must be identified as a derived requirement. Requirements must exist for both normal-range and abnormal-range (*i.e.* robustness) inputs and conditions. This helps ensure the software will continue to operate correctly to some extent in the face of anomalies. For instance, on the one hand, a normal-range requirement will describe the actuation of a valve when a hydraulic circuit is pressurized to a value within a given range. On the other hand, a robustness requirement will describe the software's behaviour if the pressure value exceeds the maximum defined operational pressure for the hydraulic circuit.

DO-178C acknowledges that errors may be introduced at any moment during the development of data items. Hence, V&V is a transverse process responsible for detecting and reporting errors that may have been introduced during any process. Detecting and reporting errors warrants a set of specific objectives. For this, V&V must perform a combination of reviews, analyses and testing over the processes' outputs. Reviews and analyses of the outputs from the requirements and design processes must ensure that HLRs are traceable to SRATS, LLRs are traceable to HLRs, and HLRs and LLRs exist for both normal-range and robustness inputs and conditions. Testing involves the creation of specific test cases from the HLRs to test software responses to normal-range and robustness inputs and conditions. In particular for the verification of the outputs from the design process, some of DO-178C's objectives our industry partners must achieve are: 1) the detailed design is accurate and consistent, 2) the detailed design conforms to design standards, 3) the architectural design is consistent, and 4) the architectural design conforms to design standards. Consistency in the previous objectives making reference internal consistency of the design where a modelled element appears in more than one design

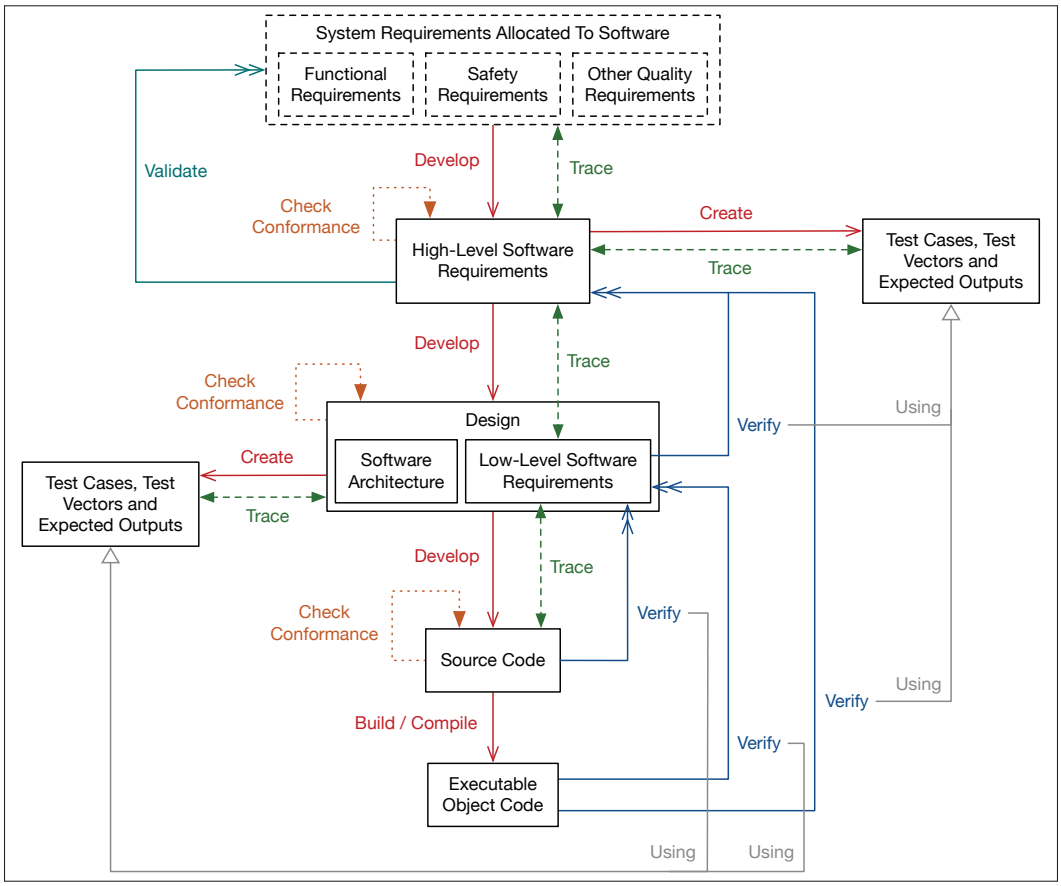


Figure 1.1 DO-178C detailed software development workflow.

model. In such a case, the element must exhibit the same properties and behaviour in all of its occurrences in the design models.

DO-178C maintains the core text of its predecessor, DO-178B, with some needed corrections to errors and inconsistencies, and improvements to interpretability, but most importantly, it addresses new considerations and practices in key areas of contemporary software development. These areas are MDE, OOT and formal methods. The new considerations and practices have been produced and published as separate, supplement documents (DO-331, DO-332 and DO-333, respectively) referring to, or modifying content of the main DO-178C document where applicable, as well as including additional terminology and content where necessary. The focus of the industry partners was on using MDE technologies to support their development and certification efforts. Thus, the DO-331 supplement became applicable. Although the DO-332

supplement is particularly intended for the software coding and integration processes, compliance with its guidance also becomes relevant when using object-oriented modelling languages like UML, which provide constructs for representing OOT features, *e.g.*, inheritance, polymorphism, overloading. In the following subsections we briefly present DO-331 and DO-332.

1.1.2 DO-331

DO-331 (Rad, 2011b) is a supplement to DO-178C containing development and verification guidelines when parts of the software are to be expressed using models. Selic (2003) defines a model as a reduced (*i.e.* simplified, abstract) representation of some (aspect of a) system that highlights its properties of interest from a given viewpoint. With DO-331, models may be used for software analysis, verification, simulation or code generation. Two types of models are addressed: *specification models* and *design models*. The former represent HLRs and the latter represent LLRs and/or software architecture. Even though DO-331 supports the use of models, it does not alleviate the objectives defined in the main document, therefore, as models represent requirements and/or software design they must be treated as such. For instance, the verification of design models must, therefore, satisfy the four objectives mentioned previously. Traceability between models and code is also required; code derived from a model must trace back to a requirement.

Figure 1.2 shows the scope of the specific guidance provided by DO-331 within DO-178C. Models may coexist alongside textual requirements specifications. Compliance with the guidance of this supplement is mandatory whenever models are used in the life cycle. Coverage analysis is meant to be performed in a similar way as with textual requirements since models are to be treated as equal. All requirements contained in the model(s) must be exercised by test cases and all model structures must be developed from the requirements.

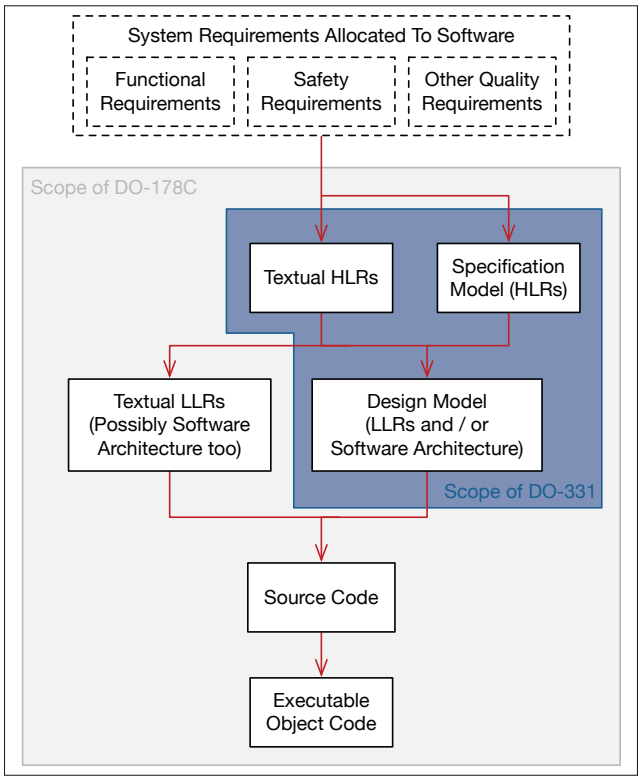


Figure 1.2 Scope of DO-331 within DO-178C. Adapted from Rad (2011b).

1.1.3 DO-332

Object-oriented technologies (OOTs) have existed since before the 1970s, prior to the publication of DO-178's first revision. However, at that time OOTs were just gaining attention and were not in common, universal usage as they are today. This situation led to the creation of DO-332 (Rad, 2011c). DO-332 contains compliance needs when the software—or parts of it—are to be coded using object-oriented programming languages and taking advantage of all the features that have made them so popular, *e.g.*, inheritance, polymorphism, overloading, type conversion, exception management, dynamic memory management, and virtualization.

DO-332 has various modifications to the main guidelines in DO-178C due to the shift in the programming paradigm from procedural languages. OOTs are founded on the principle of strong abstraction where the programming unit, an *object* (defined by a *class*), holds both the

data and the *methods* (or procedures) to manipulate such data. Class hierarchies and the interactions among the instantiated objects are what delivers the expected functionality of the system. DO-332 includes specific development compliance needs covering class hierarchies. In particular, DO-332 suggests class hierarchies should be derived from HLRs, and bidirectional traces should be defined between HLRs and methods, since it is in the latter where the former are implemented.

Regarding verification, DO-332 calls for the verification of, among other things, class hierarchies for consistency with HLRs, and local type consistency. Normal-range and robustness testing should still be performed.

1.2 Model-Driven Engineering (MDE)

Models, as defined by Selic (2003), are reduced (*i.e.* simplified, abstract) representations of some (aspect of a) system that highlights its properties of interest from a given viewpoint. Model-Driven Engineering (MDE) was developed around the premise to turn models into first-class artifacts across the entire system development life cycle in an attempt to deliver higher quality systems in the most productive way possible while reducing their complexity (Schmidt, 2006; Bézivin, 2006). In order to keep up with such a premise, MDE makes use of domain-specific modelling languages (DSMLs) to provide users with a working environment where they can directly manipulate domain concepts (Kelly & Tolvanen, 2007).

1.2.1 Modelling

Models are created by using some *modelling language*. Modelling languages share the common properties of languages (Kurtev *et al.*, 2006; Arboleda & Royer, 2012), *i.e.* they have: 1) a *concrete syntax* or notation for the construction of models, 2) an *abstract syntax* or vocabulary of the domain concepts they represent, 3) mappings between the abstract and concrete syntaxes, and 4) an implicit or explicit *semantics* or the way to create well-formed models with the defined domain concepts. These properties are usually defined through a metamodel. A

metamodel is a model with a higher abstraction level describing the domain concepts, their relationships and the structural constraints that guide their combination. The relationship between a model and its defining metamodel is called *conformance* (Bézivin, 2005). Since metamodels are also models, they need their own modelling language in order to be described. This language is represented by a meta-metamodel. This definition may continue *ad infinitum*, hence, the Object Management Group (OMG) introduced its model-driven architecture as a four-level modelling framework where the highest level contains a unique modelling language called the Meta-Object Facility (MOF). MOF defines a meta-metamodel that is defined by itself.

Figure 1.3 illustrates the modelling levels of the OMG's MOF framework (OMG, 2013). The best-known example of a metamodel (or M2 model) is UML, of which MOF is its meta-metamodel. An M0 model represents the data that is to be described. The semantics, nonetheless, is sometimes more difficult to express through a metamodel alone and usually requires a helper, formal notation in order to be achieved. For this purpose, the MOF framework uses the Object Constraint Language (OCL) in modelling languages defined within its context. In the case of UML, its defined modelling language may be extended through what are called profiles, which are particular constructs (*i.e.* abstract syntax elements, their mappings to UML's concrete syntax, and semantics) that augment UML for its use in certain domains.

There is a variety of system aspects that models may represent. Furthermore, models may be used by different stakeholders. Because of these situations, models can be classified in terms of two dimensions: 1) the system perspectives or views offered for the system, and 2) their level of abstraction. Views make reference to the different representations that models may portray of how aspects of a system are addressed (Rozanski & Woods, 2011). An individual view is intended to describe a separate aspect of a system, but when taken collectively, the views are intended to provide a description of the entire system. Some examples of common views are: functional structure, layering, inter-component communication, deployment. Levels of abstraction make reference to the amount of implementation details that models are able to capture. The higher the level of abstraction a model has, the less implementation details it can represent and, thus, it will be found to be closer to the problem's space. Models with high

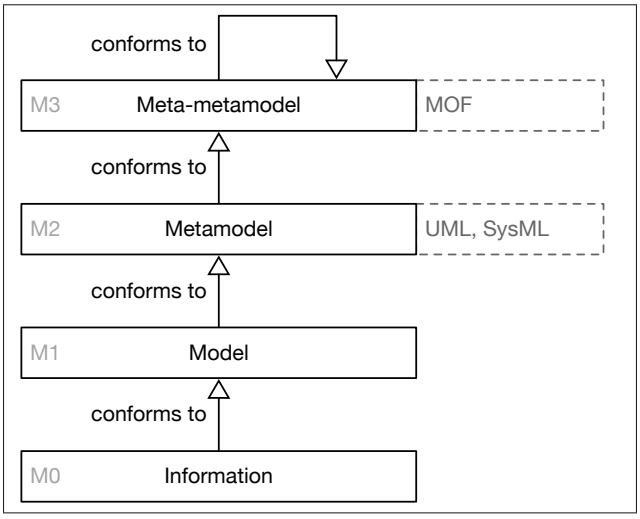


Figure 1.3 OMG's four-level modelling framework. Adapted from OMG (2013).

levels of abstraction are said to be platform-independent models (PIMs) since their lack of implementation details render them technology free. On the contrary, the lower the level of abstraction a model has, the more implementation details it can represent and, thus, it will be found to be closer to the solution's space. Models with lower levels of abstraction are said to be platform-specific models (PSMs) since the inclusion of implementation details tailor them to specific systems, infrastructures, platforms or technologies. The following subsections will glance at how modelling is done in the set of modelling languages used by the industry partners and then present a methodology to build custom DSMLs for regulation certification.

1.2.2 Modelling with Simulink and Stateflow

Simulink (MathWorks, 2018a) is a graphical block modelling language with predefined and customizable blocks for describing continuous causal relationships between blocks that determine system behaviour (see Figure 1.4). The interconnection of blocks expresses calculation procedures and not the actual system structure. Nevertheless, logical structures can be imposed by establishing a hierarchy of such blocks. Stateflow (MathWorks, 2018b) extends Simulink with a state-like formalism variant of Harel's Statecharts. Stateflow enables the representation

of control functions that are dependent on a combination of past and present logical conditions (see Figure 1.5). Since Stateflow is an extension to Simulink, Stateflow models (*a.k.a.* Stateflow charts) must reside within a Simulink model.

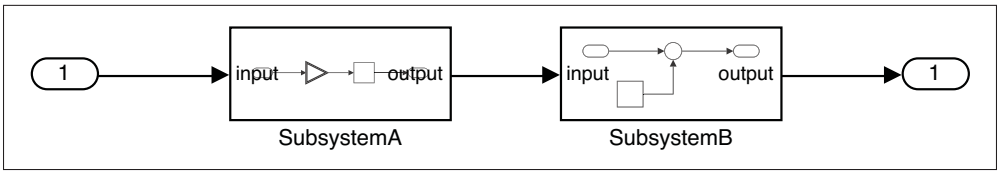


Figure 1.4 Example of the Simulink notation. Extracted from Paz *et al.* (2020).

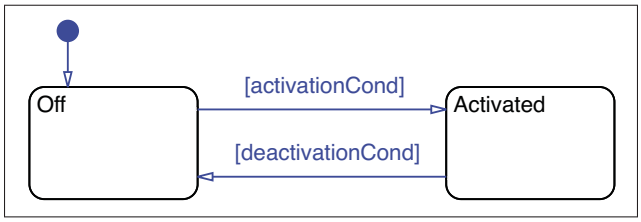


Figure 1.5 Example of the Stateflow notation. Extracted from Paz *et al.* (2020).

The MathWorks Automotive Advisory Board (MAAB), a group of major automotive OEMs (Original Equipment Manufacturers) and suppliers, created the Control Algorithm Modeling Guidelines using MATLAB, Simulink, and Stateflow (MathWorks Automotive Advisory Board, 2012). The MAAB guidelines are intended to enable model exchange and facilitate the use of the MathWorks' tools for MDE. Initially, they were intended for use within the automotive industry but have been adopted by other major industries building safety-critical systems, like avionics (Erkkinen, 2005).

Simulink models under the MAAB guidelines are mandated to be organized in a hierarchical, three-layered structure. The top layer describes a control system as a single block performing a function over a set of input variables to produce a set of output variables. The structure layer describes the controller of the previous layer as a decomposition of interconnected atomic subsystem blocks (see Figure 1.4). Stateflow charts may be included as part of this layer if

modal logic is required to control a function. Lastly, the data flow layer describes each of the atomic subsystems in the previous layer as a decomposition of interconnected basic blocks (*i.e.* those from the Simulink base library, *e.g.*, arithmetic operators, logical operators, relational operators) (see the inside previews of the subsystems in Figure 1.4). Stateflow charts may be included as part of this layer as well, if modal logic is required to control a function.

1.2.3 Modelling with AADL

AADL (SAE, 2017) is a standardized textual and graphical modelling language for describing a system architecture and its runtime environment in terms of its constituting components. A component is characterized by five elements (Feiler *et al.*, 2006): 1) identity (*i.e.* type and name), 2) interfaces with other components, 3) distinguishing properties, 4) subcomponents, and 5) interactions between its subcomponents. The dynamics of components can be described by annotating them with references to other models in other modelling languages (*e.g.*, Simulink, Stateflow).

AADL includes three groups of components: software (process, thread, thread group, subprogram and data), hardware (device, processor, memory and bus) and system. Listing 1.1 present a sample system component made up of three devices and whose dynamics are controlled by a software process. Figure 1.6 presents the graphical equivalent. AADL models initially represent a high-level architecture through system, process and device components, and their interactions. Specific quality attributes (*e.g.*, timeliness, fault-tolerance, security) can be provided to the components as part of such an architectural description. From this representation, subsequent modelling steps further detail the composition of each high-level component in much the same way.

1.2.4 Modelling with UML

UML is a standardized general-purpose graphical modelling language for describing a software system (OMG, 2017b). It is capable of capturing both the structure and the dynamic behaviour

Listing 1.1: Example of the AADL textual notation.

```
system implementation sc_system.sample_system
 subcomponents
   sample_sensor: device sensor_device.sample_sensor;
   sample_interface: device user_interface_device.sample_interface;
   sample_actuator: device actuator_device.sample_actuator;
   sample_process: process sc_process.sample_process;
 connections
   sensor_data_connection: port sample_sensor.sensor_data -> sample_process.
        sensor_data;
   set_parameter_connection: port sample_interface.set_parameter -> sample_process.
        set_parameter;
   actuation_command_connection: port sample_process.actuation_command ->
        sample_actuator.command;
end sc_system.sample_system;
```

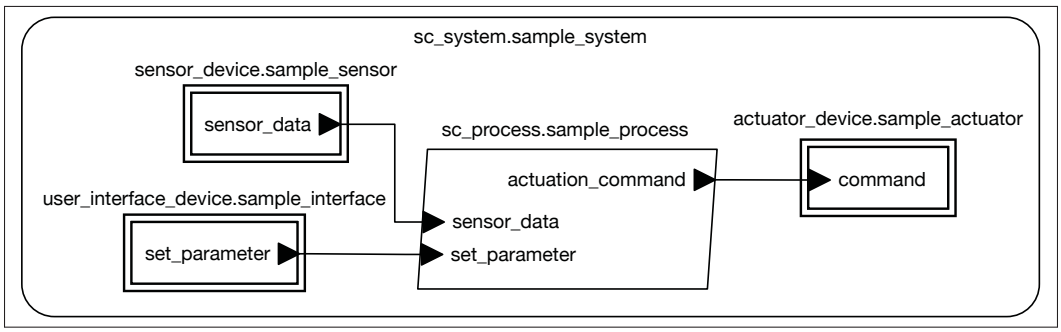


Figure 1.6 AADL graphical notation for the example in Listing 1.1.

of the system. Structure is captured as a decomposition of connected discrete objects whose collective work performs the desired system function. Dynamic behaviour is defined as the history of such objects over time as they interact with one another to perform the desired system function. UML structural constructs, like components and classes, and their interconnection express acausal relationships, meaning there is no predefined direction for such interconnections. The definition of the calculation procedure is left for the behavioural constructs. UML's behavioural semantics is based on a framework that only deals with event-driven or discrete behaviour. However, the interval of time separating two events is left unspecified in the standard. As a result, it can be considered to be as small as needed, which would allow the description of continuous behaviour (OMG, 2017b).

Avionics systems manufacturers, with their heavy reliance on standards, see in UML long term sustainability and interoperability (Le Sergent *et al.*, 2016). Indeed, popular modelling tools among them, like SCADE Suite, are UML-based (Le Sergent *et al.*, 2016). Moreover, research has been carried out in the domain to propose modelling approaches supported by UML (*e.g.*, Stallbaum & Rzepka (2010); Zoughbi *et al.* (2011); Biggs *et al.* (2016)).

Modelling with UML is somewhat arbitrary as the language and its specification allows to exercise more freedom in a design than, for instance, Simulink and the MAAB guidelines permit. Nonetheless, a hierarchical, 3-layered structure can be followed in an analogous way as with Simulink and Stateflow to simplify syntactical and semantical comparisons between them. The top layer describes the software as a modular, reusable component providing its specified functionality through its set of provided interfaces. The component can use other components by requiring any provided interface. The middle layer decomposes the software into a set of modular, reusable components providing their functionality through their sets of provided interfaces. The components may as well use each other by requiring any of the provided interfaces. Lastly, the bottom layer describes each of the components' implementation of their provided interfaces with one or more classes. State machines can be used to capture the internal state transitions of the realizing classes.

1.2.5 Methodology for developing modelling languages for regulation certification using UML

As mentioned previously, UML supports the development of domain-specific modelling languages (DSMLs) through its profile mechanism, which comprises semantic variation points and special language constructs intended for refinement (*e.g.*, Stereotypes) (Selic, 2007). Selic (2007) proposed a systematic approach with the purpose of guiding the development of UML profiles that are technically valid (*i.e.* do not contravene the UML standard) and of good quality. The general flow of the approach is: 1) develop the conceptual model of the domain (or domain metamodel), and 2) map its concepts to elements in the UML metamodel. Depending on the complexity of the DSML there may be the need to iterate a few times over the process

to ensure conformance with UML. Panesar-Walawege *et al.* (2013) refined this systematic approach to provide specific guidance for developing DSMLs for the management and collection of information for certification. The domain metamodel should result from a careful qualitative analysis of the contents of a given regulatory guideline. The UML profile and its associated OCL constraints are then created based on the resulting domain metamodel.

Figure 1.7 illustrates the methodology as presented by Metayer *et al.* (2019). The domain metamodel specifies what will be represented with the DSML and how. It is strictly defined considering the needs of the domain (*i.e.* the regulatory guideline) without concerns about the UML metamodel. Four elements make up the domain metamodel: 1) the set of fundamental language constructs, 2) the set of relationships existing between the previous constructs, 3) the set of constraints for the production of valid models, and 4) the semantics of the language. Mapping the domain metamodel to the UML metamodel goes through all the domain concepts and identifies the most suitable UML metaclass for each one. A domain concept is defined as a stereotype to be applied over a selected UML metaclass. The selection of the UML metaclass is done by identifying the metaclass with a semantics closest to that of the domain concept, and checking that the constraints of the selected metaclass and those of its superclasses do not conflict with the constraints of the domain concept. If necessary, refinements to the attributes of the selected metaclass are performed. The same is done for the associations of the selected metaclass to other metaclasses. If there are conflicting associations, these can be constrained with the constraints defined for the stereotype.

1.3 Model-Based Approaches Supporting Safety-Critical System Development and Certification

Models have been pushing their way into safety-critical systems development and certification since around the mid-1960s when computer-aided design first appeared. Despite being particularly interested in avionics software and its development and certification under DO-178C, we extended our review scope of existing model-based approaches to include safety-critical software in general. This is because safety-critical software share various characteristics across

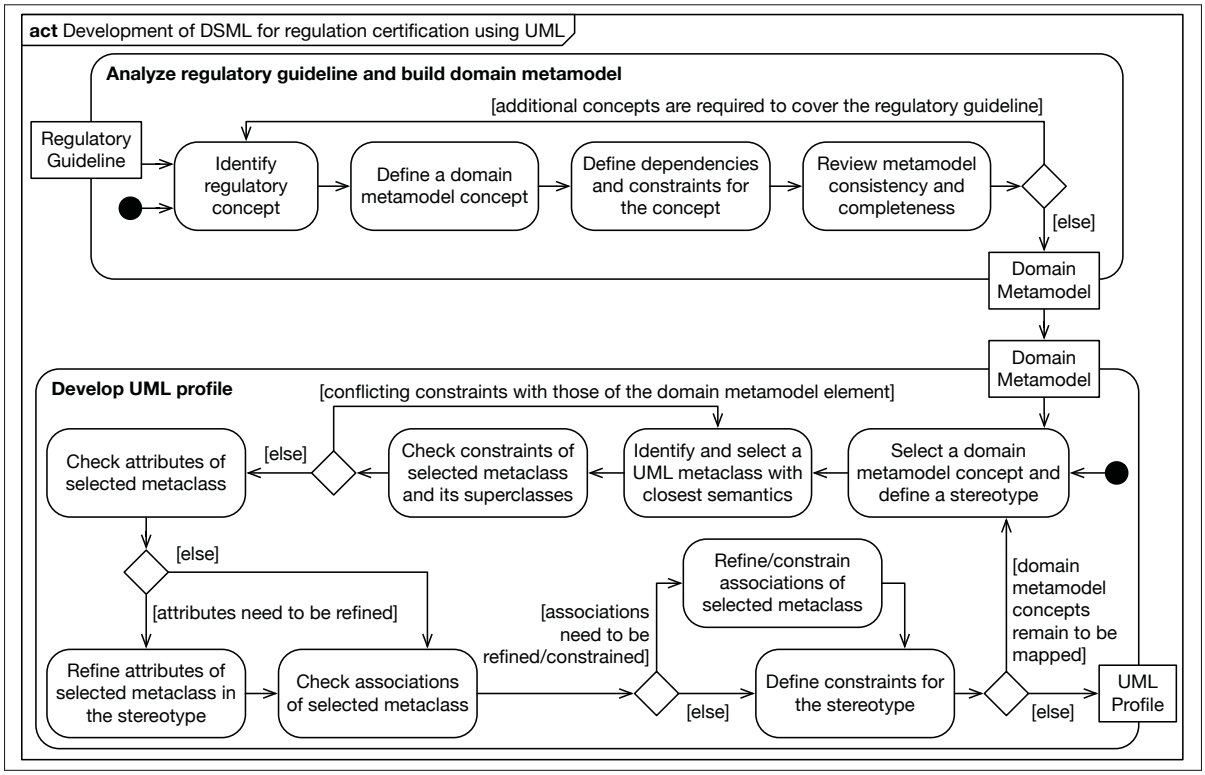


Figure 1.7 Methodology for developing UML-based DSMLs for regulation certification. Extracted from Metayer *et al.* (2019).

other domains where they are also deployed, such as railway, medical, maritime and energy. Furthermore, relevant model-based approaches are found to be labeled as domain-independent solutions.

In recent years, Feiler *et al.* (2006), Stallbaum & Rzepka (2010), Blouin *et al.* (2011), Zoughbi *et al.* (2011), Panesar-Walawege *et al.* (2013), Wu *et al.* (2015) and de la Vara *et al.* (2016) have proposed several model-based approaches. There are also toolchains based on proprietary modelling languages and tools, such as those from MathWorks (Eisemann, 2016). Existing approaches can be divided into three groups: 1) approaches supporting certification, 2) approaches supporting requirements specifications in the context of certifiable safety-critical systems development, and 3) approaches handling design model heterogeneity. The following subsections present synthesized descriptions of several approaches in each group.

1.3.1 Approaches supporting certification

We studied existing MDE approaches in terms of their support to produce and certify avionics software under DO-178C. The study has been published as part of the proceedings of the International Symposium on Software Reliability Engineering (ISSRE 2016) Workshops (Paz & El Boussaidi, 2016). To carry out the assessment we built a framework to characterize and compare the selected approaches according with several criteria. We defined the following four groups of criteria: 1) their objectives and targeted stakeholders, 2) the extent to which they support DO-178C guidelines, 3) the way they handle and present information, and 4) the extent to which the approach is ready for use. The documentation of our characterization framework can be found in Appendix I. The reviewed approaches can be divided into those approaches that are meta-approaches, those for safety assurance cases, and those for certification-compliant system design and analysis.

1.3.1.1 Meta-approaches

Meta-approaches take a different perspective on model-based support for development and certification. Their goal is to propose a generic way to build models in a sense that every model can be considered as the starting point of an approach on its own. Panesar-Walawege *et al.* (2013) suggest a methodological approach for guiding the management and collection of information for certification. They define a four-step process based on Product Line Engineering (PLE) principles, summarized in Figure 1.8. Such a process starts by a careful qualitative data analysis of the contents of a given regulatory guideline or standard to create a conceptual model of such a guideline or standard. The objective of the conceptual model is to aid suppliers/manufacturers in creating the evidence necessary for certification according with the given standard. A UML profile and its associated OCL constraints are then created based on the conceptual model of the given regulatory guideline or standard and their relationships with application domain concepts.

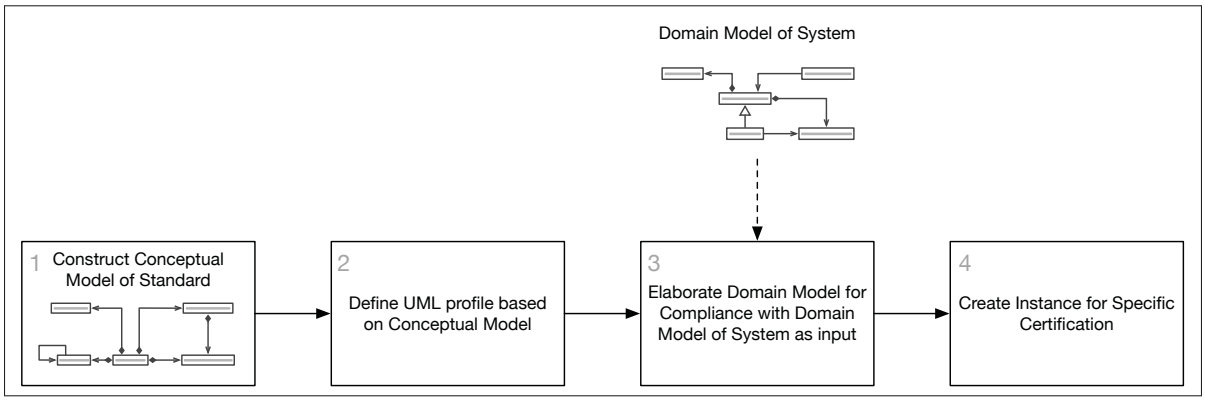


Figure 1.8 Process for managing and collecting certification information. Adapted from Panesar-Walawege *et al.* (2013).

Other approaches were developed in the context of a European-wide project, the OPENCOSS project, to produce an open safety certification platform. The OPENCOSS project had two main goals. On the one hand, reduce the recurring costs of safety (re-)certification. On the other hand, increase product safety. To achieve these goals the project 1) proposed a common certification framework that facilitates reuse of assurance artifacts within and across domains such as railway, avionics and automotive, and 2) established an open-source safety certification infrastructure. The works of Luo *et al.* (2013), de la Vara *et al.* (2016) and Ruiz *et al.* (2016) contribute to the OPENCOSS project by introducing more systematic certification practices based on the modelling of safety compliance needs.

Luo *et al.* (2013) propose an approach to systematically and objectively model safety standards. The resulting models can then be used for demonstrating compliance with the safety standards and for enabling reuse of the developed assurance artifacts. The approach involves the creation of three types of models: *structure model*, *conceptual model* and *process model*. The structure and conceptual models help achieve an unambiguous understanding of the safety standards, the latter playing the role of guideline when complying to the safety standard. The process model can be used in demonstrating compliance of the project's process with the process described by the standards.

Ruiz *et al.* (2016) develop the *Common Certification Framework*, an approach that assists on the systematic reuse of compliance justifications and safety certification artifacts across standards and domains. The Common Certification Framework consists of several languages or metamodels allowing the representation of safety-related information from two main sources: the standard for which compliance must be demonstrated and the product for which compliance is sought. The Reference Assurance Framework (RAF) Metamodel is intended for supporting the specification of safety compliance needs that have or might have to be considered.

de la Vara *et al.* (2016) present in greater detail the RAF metamodel. The RAF metamodel includes the necessary concepts and relationships to tailor safety compliance needs models to project-specific characteristics. Safety compliance needs may come, for example, in the form of compliance needs to fulfill, artifacts to manage or activities to execute, and from multiple sources such as specific safety standards, recommended practices or company-specific practices. A *baseline* model will hold the specific compliance needs a project should provide assurance for. This baseline model typically contains a subset of the compliance needs present in a particular targeted standard and be complemented with additional needs that are specific to the project's characteristics. Project-specific aspects can be captured with a set of metamodels built for such purposes. The *process* metamodel captures the process that is to be executed to create a product. The *evidence* metamodel captures evidences of safety and of compliance. The *argumentation* metamodel captures arguments used to justify the safety-related decisions that are taken throughout the project. A *vocabulary* metamodel captures the items of vocabulary for naming and describing safety assurance terms and concepts. The items of vocabulary may come from the text of safety standards or be company-, product- or application-specific. The *mappings* metamodel serves to specify the degree of equivalence between models conforming to the previous metamodels.

1.3.1.2 Safety assurance cases

Safety assurance is a fundamental element in safety-critical system certification. Assurance cases have been in use for such a purpose for a long time (Hawkins *et al.*, 2013). An assurance

case is a model capable of communicating a clear, comprehensible and defensible argument that particular requirements have been satisfied to deem the system safe to operate in a particular context (Hawkins *et al.*, 2013; OMG, 2018). The model consists of a set of auditable claims as well as arguments and evidence developed to support such claims (OMG, 2018). The assurance case should additionally describe the system's scope and its operational context. The model starts by defining some high-level *claims* (also referred to as *goals*) and braking them down into lower-level claims until they can be supported by *evidence*. Claims are stated within a *context*. Rationales are captured as *assumptions* and *justifications*. Complex assurance cases may be created by interrelating separate *modules* of arguments.

Hawkins *et al.* (2013) provide a comparative examination of assurance cases and compliance towards DO-178C. They had two objectives. On the one hand, to highlight the advantages and limitations of assurance cases and the traditional compliance of DO-178C. On the other hand, to describe the role assurance cases can play when seeking compliance with DO-178C. In the context of providing assurance for a safety-critical avionics system, Hawkins *et al.* (2013) propose to construct an assurance case following these four principles: 1) safety requirements shall be defined to address the system's contribution to hazards, 2) the intent of the safety requirements shall be maintained throughout requirements decomposition, 3) safety requirements shall be satisfied, and 4) hazardous behavior shall be identified and mitigated.

The OMG (2018) developed the Structured Assurance Case Metamodel (SACM) to standardize the way of presenting and structuring arguments in an assurance case. The SACM is an effort to promote the use of assurance cases as a way to help the assurance of safety and its reuse within and across projects. The SACM specification is divided into three metamodels: the *argumentation* metamodel, the *artefact* metamodel and the *terminology* metamodel. The rationale behind this division is to allow the individual exchange of argumentation and evidence while being able to use them in combination. The *argumentation* metamodel defines the constructs for building and interchanging structured argumentation statements. The *artefact* metamodel defines the constructs for building and interchanging evidence-related statements, such as describing artifacts (*i.e.* their properties and associated events), collecting and man-

aging evidence (*i.e.* relationships between participants, activities, resources and associated evidence artifacts), and structuring of artifacts (*i.e.* artifact composition). The *terminology* metamodel defines constructs that can be used to represent compliance needs associated with an assurance standard, or project- or system-specific characteristics. However, no relationships exists between the concepts of the *terminology* metamodel to those in the *argument* metamodel and the *artefact* metamodel.

1.3.1.3 Certification-compliant system design and analysis

Berkenkötter & Hannemann (2006) propose a UML profile for modelling critical railway control applications. The approach is intended to connect railway engineering practitioners with the development techniques of safety-critical software. The profile captures the physical elements of railway control systems (*e.g.*, sensors, signals, routes, crossings, points, segments), their physical topology and the composition of the controller's operations. Figure 1.9 shows an object diagram representing two sensors on the connection of two bidirectional tramway track segments. The left side of the figure shows the notation of the network elements and their relationships as seen by a railway engineering practitioner. The right side of the figure shows the corresponding instance stereotype for each of the modelling elements and relationships. The UML profile is complemented by a formal (mathematical) behavioural model. From the instance model, and based on the formal behavioural model, automated code can be generated for railway controller systems and for safety compliance verification tasks.

Wu *et al.* (2015) proposed a modelling language for the architectural design of avionics systems in accordance with DO-178C. The modelling language's semantics is based on a set of *safety properties*, *i.e.* requirements derived from the objectives and activities defined in DO-178C (*e.g.*, "A safety component and its safety interfaces have the same assurance level."). Their approach focuses on modelling system components that have a set of associated safety concerns and their interfaces. Components may interact with one another via a safety channel. In addition, components should provide a way to detect and handle faults.

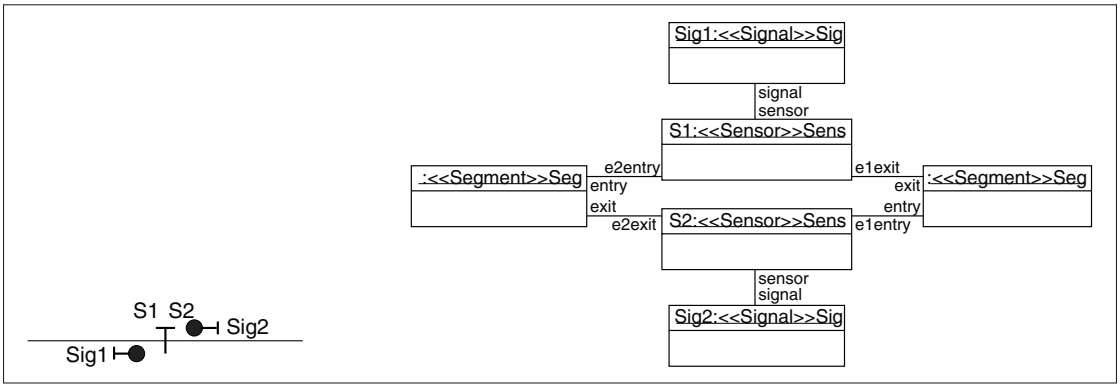


Figure 1.9 Fragment of an object diagram for a tramway network design. Extracted from Berkenkötter & Hannemann (2006).

Graf *et al.* (2006) and Hooman *et al.* (2007) developed OMEGA-RT, a UML profile for modelling real-time embedded systems. It extends UML, particularly class diagrams and state machine diagrams, with the ability to capture time- and scheduling-related data. Figure 1.10 gives an example state machine diagram for the possible states of a sensor in a flight control system. The transitions between the states are constrained by OMEGA-RT timed annotations, which are time stamped UML events. The `timeout(timeglobal)` event is an example of a time stamped event. The aim of OMEGA-RT is to support the analysis and verification of the time and scheduling-related properties characteristic of real-time embedded systems. On the one hand, OMEGA-RT represents any operational concepts as classes with the different kinds of object-oriented structures and associations corresponding to them (*e.g.*, polymorphism, inheritance, aggregation) constrained by events described using OCL. On the other hand, behaviour of such classes is captured in state machine diagrams annotated with time and duration information.

Stallbaum & Rzepka (2010) contribute a modelling approach targeted towards the specification of test models that can serve both for testing and as supporting evidence in a certification process. The approach consists of a UML profile to extend UML behaviour diagrams, particularly activity diagrams, with safety-related information relevant for DO-178B compliance. Each model element is given a stereotype allowing the definition of test and certification-relevant information. With this formalism, Stallbaum & Rzepka (2010) focus on including

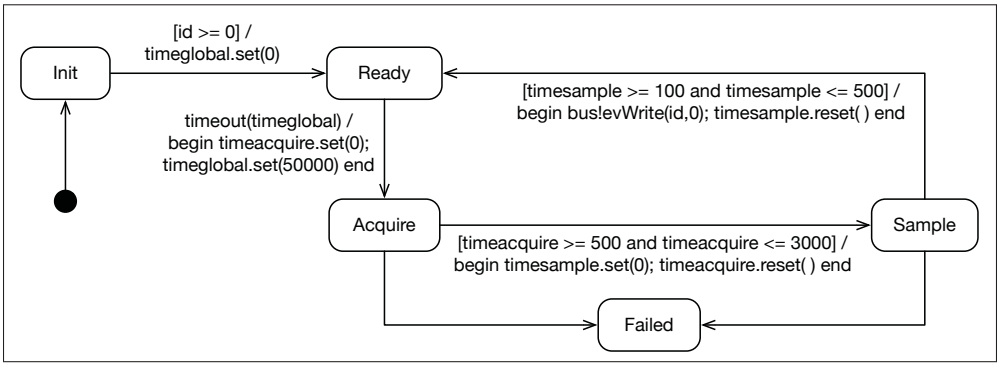


Figure 1.10 State machine diagram using OMEGA-RT for an aircraft's flight control computer sensor. Adapted from IST (2001).

constructs for eight essential testing needs specified in DO-178B: 1) traceability links from test model elements to requirements, 2) software levels and their rationale, 3) normal and abnormal test conditions in test cases, 4) testing method (*e.g.*, hardware-software integration, software integration), 5) hardware/software interfaces and their parameters, 6) traceability links from test model elements to software components, 7) traceability links from test model elements to source code, and 8) types of traceability links.

Eisemann (2016) describes a toolchain for DO-178C-compliant avionics software development. The toolchain is composed of tools from various well-known and well-established tool vendors. Among the main components of the toolchain are MathWorks Simulink and dSPACE TargetLink providing the modelling environment for creating both specification models and design models according with DO-331 as well as the infrastructure for automatic code generation. Another main component of the toolchain is BTC EmbeddedTester, which covers various verification objectives of DO-331 via model-based testing. BTC EmbeddedTester is capable of automatically creating requirements-based test cases and executing them over the Simulink models. Figure 1.11 maps the use of all these tools to the DO-178C reference workflow of Figure 1.1. HLRs can be specified in the form of specification models with Simulink. Requirements-based test cases can be automatically generated from those Simulink models using BTC EmbeddedTester. Specification models in Simulink can drive the software design to produce design models in Simulink. BTC EmbeddedTester can again take these Simulink

design models and automatically generate test cases. Automatically generating source code is possible with the Simulink environment in order to transform Simulink design models into ANSI C code. BTC EmbeddedTester comes into play to automatically execute the generated test cases over both the design models and the implementation code.

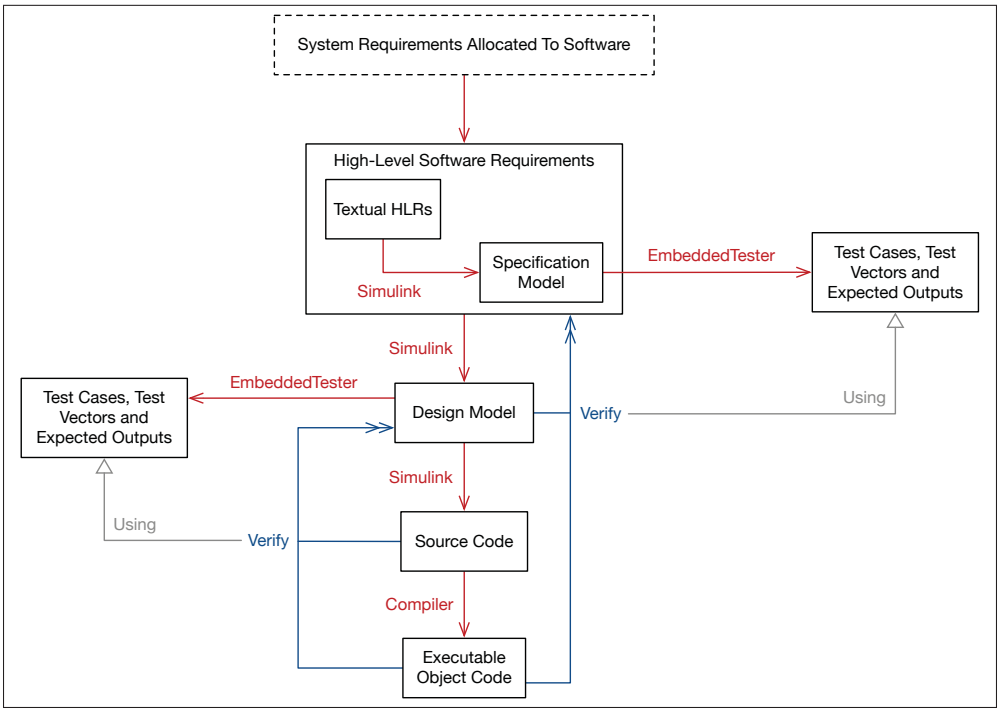


Figure 1.11 Toolchain based on commercially available tools. Adapted from Eisemann (2016).

1.3.2 Approaches supporting requirements specifications

Several requirements modelling languages were proposed for safety-critical systems development. They can be divided into three categories: 1) those using natural language-based specification, 2) those providing a formal notation, mostly through behavioural modelling, and 3) those offering a hybrid specification.

1.3.2.1 Natural language-based specification

SysML is a standardized general-purpose graphical modelling language (OMG, 2017a). Although it was introduced for system engineering, it has been widely used for safety-critical system engineering by practitioners and researchers (*e.g.*, (Sakairi *et al.*, 2012; Biggs *et al.*, 2016)). SysML, with its requirements diagram, captures natural language requirement statements and their relationships to other requirements (*e.g.*, indicating hierarchical level, copy, derive) and to other elements as well (*e.g.*, test cases and design blocks). The main structural concept for requirements modelling with SysML is the `Requirement` stereotype, which includes properties to capture a unique identifier and the text of the requirement. Verification status, priority and other additional properties can also be specified. SysML allows extending the `Requirement` stereotype to represent requirements taxonomies (*e.g.*, functional, interface, performance). A hierarchical tree-like structure of packages can also be defined to contain and organize all of the requirements. Several types of relationships exist to represent possible associations of requirements with other elements: *composition*, *derive*, *refine*, *satisfy*, *verify*, and *trace*. The *composition* relationship creates a hierarchy of requirements where a requirement can be decomposed into simpler child requirements at the same flowdown level, which can be easier to both satisfy and verify. The *derive* (`deriveReqt`) relationship associates a requirement with another requirement with the intent of imposing additional considerations and constraints based on further analysis of the system. The *refine* relationship can exist between a requirement and any other element (can be from a different model) or vice versa. This relationship expresses how a model element further refines the other model element. The *satisfy* relationship is used to describe how design (or implementation) elements satisfy the requirements. This association, however, is not intended to constitute a proof that the requirement is indeed satisfied by the given element, this is the objective of the *verify* relationship. The *verify* relationship can associate a model element, such as a `testCase`, to verify a requirement. Finally, the *trace* relationship is a general-purpose relationship to associate a requirement with any other element, other requirements included.

Zoughbi *et al.* (2011) propose a UML profile, based on concepts found in DO-178B, for capturing natural language safety-related requirements allocated to software and enabling the monitoring of the design and implementation of the software with regard to the provided safety requirements. The concepts in this profile are partitioned into five packages (Figure 1.12 depicts the structure of the first three): 1) the requirements package, comprised of the necessary concepts to capture requirements and their refinement as well as the traceability links to code artifacts and the design rationales, 2) the characteristics package, containing concepts that capture the elements with a direct impact on safety, identify their respective software level and associate the design strategies to be followed, 3) the event management package, defining the concepts that represent events or actions with an impact on safety and the way the system should handle them to ensure safety, 4) the configuration package, including concepts for software configuration, change control and user-modifiable software, and 5) the replication package, addressing software redundancy. The UML profile by Zoughbi *et al.* (2011), also aims to support system design with the event management package. The system is designed by identifying all the events it receives and the reactions (*e.g.*, algorithms) it performs.

Nejati *et al.* (2012) address traceability management in the requirements and design phases. They propose a modelling approach for specifying safety requirements, expressing design elements, and automatically extracting design fragments relevant for a given safety requirement. Their work focuses on three fronts: 1) how to express safety requirements, 2) how to express design, and 3) traceability from requirements to design. The proposed approach captures the requirements as statements expressed in natural language, and the design as SysML models. Traceability between requirements and design elements is provided by an information model. An algorithm extracts the fragment or "slice" of the created designs that is relevant to given safety requirements. The algorithm relies on a formal notation to denote the associations between the elements conforming to the information model. The algorithm removes the model elements that are considered irrelevant to the fulfillment of the safety requirements under analysis.

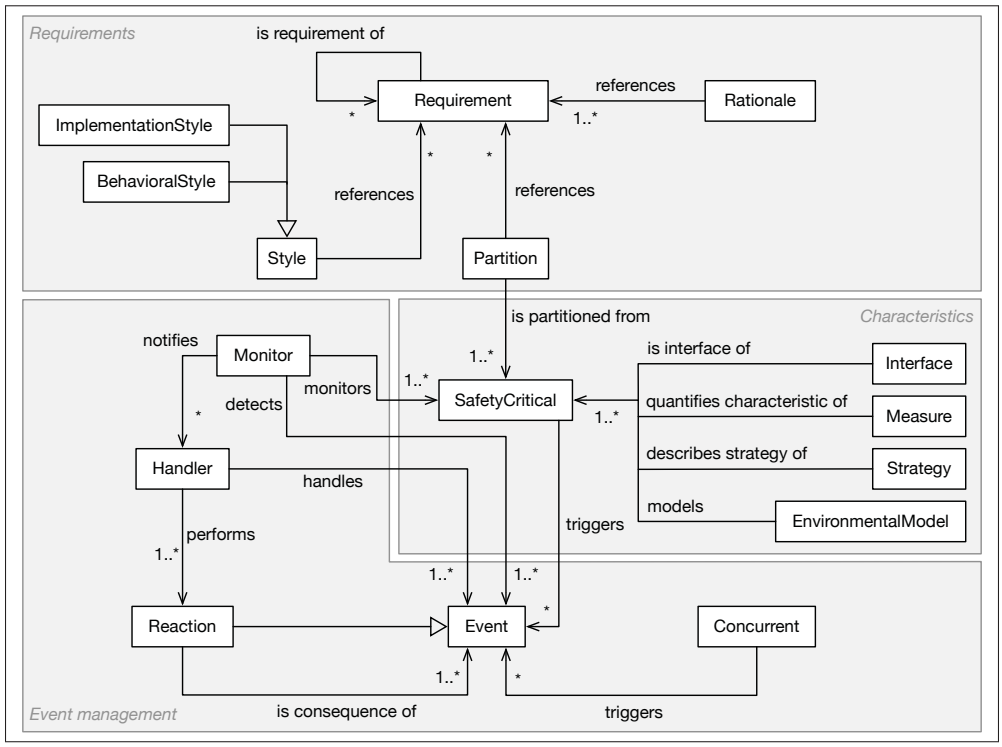


Figure 1.12 Fragment of the UML profile's metamodel by Zoughbi *et al.* (2011). Adapted from Zoughbi *et al.* (2011).

Biggs *et al.* (2016) present a SysML profile for modelling safety-related concerns of a system. They focus on the documentation of hazards for communicating them from safety engineers to system engineers in order to be considered during system design, and, conversely, for communicating from system engineers to safety engineers the hazards that the system was designed to manage. Specification is done mainly in natural language with some properties (*e.g.*, cost, priority) captured in attributes of the hazard construct. Moreover, their SysML profile includes not only constructs for software aspects but also for hardware and procedures. Thus, it is intended as a whole-system design specification modelling language.

Blouin (2013) developed the Requirements Definition and Analysis Language (RDAL). RDAL, as shown in Figure 1.13, is a modelling language for capturing, validating, analyzing and verifying system requirements. The modelling language may be used during requirements elicitation as a means of communication. RDAL has a minimalist design that focuses only on

supporting the modelling and analysis of requirements. Thus, the modelling language is meant to be coupled with other languages that support the modelling of behavioural aspects, like Use Case Maps (UCM) (Amyot, 2003), and design elements, like AADL (Feiler *et al.*, 2006). The SystemOverview concept represents the statement of how the system to be built interacts with its environment (*i.e.* a collection of contractual elements representing the external entities the system to be built interacts with). Interactions between environment entities and the system to be built are declared by means of InteractionVariables. The relationships functionUsedIn and globalSystem have the purpose of linking the requirements and system overview to the design elements that will satisfy them. The SystemOverview concept has a globalSystem property to identify the AADL system component that represents the modelling of the system to be built and its environment. AbstractRequirements are decomposed into Requirements and Assumptions. An Assumption is a special type of requirement satisfied by the environment instead of by the system to be built. A *Requirement* is expressed mainly in natural language but can be formalized using one of several constraint languages (*e.g.*, OCL, BLESS, Lute). Both the Requirement and Assumption concepts can be associated to a VerificationActivity, which is a concept that represents an activity carried out for verification such as a test case. The VerificationActivity concept can contain references to external artifacts developed in other languages (*e.g.*, Test Description Language, TDL, for modelling test cases).

ReqSpec (Feiler *et al.*, 2016) is a textual natural language requirement specification language drawn from RDAL. ReqSpec allows users to define goals (or stakeholder requirements) and requirements (or system requirements). Goals are expressed by goal declarations and requirements by requirement declarations. Goals and requirements can be organized according to the architecture structure, by associating them with AADL component types or implementations. Figure 1.14 shows examples of requirements for a system sensor specified using ReqSpec. The system requirement set declaration represents requirements for a specific architecture component and contains a set of system requirement declarations. Users can also declare a set of reusable requirement declarations through a global requirement set declaration. Such reusable requirements can then be included in system requirement set declarations.

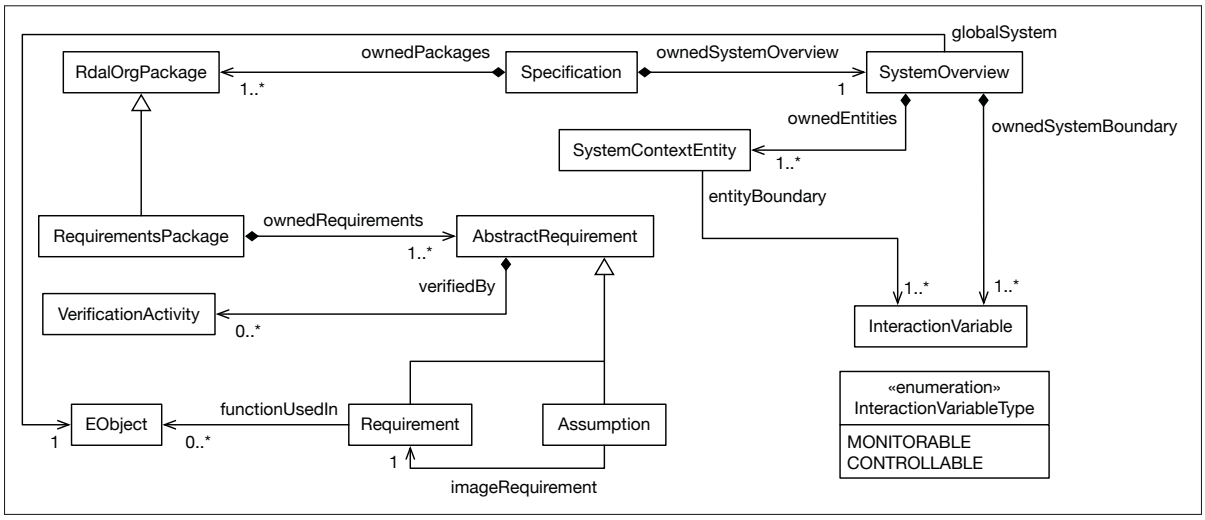


Figure 1.13 Fragment of the RDAL metamodel. Adapted from Blouin (2013).

```
system requirements PassiveSensorReqs for ASSASensors::PassiveTerrainSensor
[
  requirement Req4 : “Passive sensor”
  [
    val EnergyLevel = 0
    description “Passive sensor radiates " EnergyLevel " energy”
    value predicate #JMRMS::EnergyLEvel == EnergyLevel
    see goal MSStakeholderRequirements.SR_27
  ]
  requirements Req1 : “Spherical terrain awareness for aircrew”
  for TerrainSphere
  [
    description “Spherical SA of terrain within " DesiredObservationRadius " radius for aircrew”
    val DesiredObservationRadius = 5 nm
    compute measuredDistance ; JMRMS::NauticalDistance
    value predicate measuredDistance >= DesiredObservationRadius
    see goal MSStakeholderRequirements.SR_27
  ]
]
```

Figure 1.14 Fragment of a requirement specification for a sensor using ReqSpec. Adapted from Feiler *et al.* (2016).

1.3.2.2 Behavioural modelling

Leveson *et al.* (1994) developed RSML to model the required system's black-box behaviour including assumptions regarding the behaviour of other system components. Formal analysis procedures can be applied over the resulting model for ensuring it satisfies the system's functional and safety goals and constraints. RSML is designed for process control systems, where a

function describes the mapping between the inputs (or controlled variables) and the outputs (or manipulated variables) of the system in the face of disturbances. The function (to be computed by a controller) is specified using a state machine model. The outputs of the controller are specified with respect to state changes in the model as information is received about the current state of the controlled process via the controlled variables. Physically distinct components are modelled as separate (communicating) state machines. Inter-state machine communication is modelled as directed messages sent and received over unidirectional channels. Within a state machine, events are broadcast. Guard conditions of transitions are specified using AND/OR tables. No statement is made on the kinds of conditions that can be expressed in AND/OR tables, but seem to be limited to the following: 1) relational expressions among variables, given values and functions (*e.g.*, equalities, inequalities), 2) macros (*i.e.* reusable named and parameterized AND/OR tables), and 3) substate of a parallel state. Figure 1.15 shows an example RSML specification for the Traffic alert and Collision Avoidance System level II (TCAS II). The system specification includes a description of each input and output variable, as well as the transitions from one state to another. Manipulated variables change value after a triggering event. Also featured in this figure is RSML's hierarchical abstraction, a type of information hiding mechanism to make the specification more readable. The purpose of such an abstraction is to hide low-level information.

UCM (Amyot, 2003) is a semi-formal modelling language to discover, specify and review requirements. Requirements are described as behavioural scenarios. Figure 1.16 describes the scenario for extending an aircraft's landing gear. A *scenario* collects a set of partially ordered *responsibilities* (shown as `x`'s). A *responsibility* is something to be performed. *Scenarios* or parts of them can be allocated to *components*, a generic and abstract construct that can represent a software or a non-software entity (rectangle), which in turn can be nested within larger *components*. The scenarios progress along *paths* from a *start point* (filled circle) to an *end point* (bar). *Paths* can branch into alternative paths (*i.e. OR-fork*) upon guard *conditions* (shown between square brackets). The concept of concurrency is introduced with *AND-forks*. *Scenarios* can be integrated by using *stubs* (containers for sub-maps) to represent complex scenarios (not

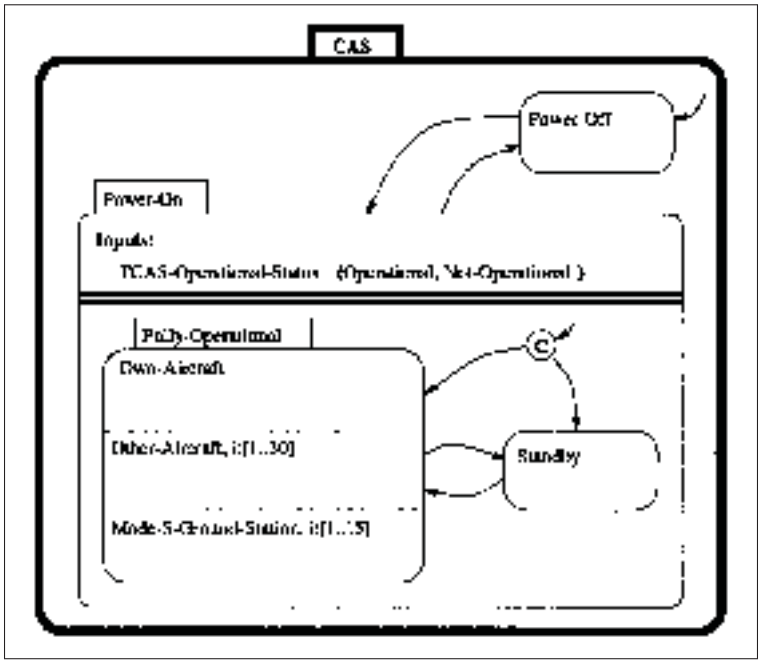


Figure 1.15 High-level RSML specification for the TCAS II. Extracted from Leveson *et al.* (1994).

shown). Waiting within *paths* can be achieved with the definition of *waiting places* (filled circle on path). *Timers* (clock icon) can be dependent on events from the environment or paths in a *scenario*. If the connected path never arrives, the *timeout* path (zigzag path) is followed.

1.3.2.3 Hybrid specification

Bitsch (2001) proposes a pattern-based approach to simplify the specification of system safety requirements when using formal languages. Patterns provide the advantage of capturing expert knowledge and experiences to help in the correct formulation of the safety requirements and in their identification. Furthermore, the approach enables formal verification to be carried out over the set of requirements. The approach consists of a catalog of pre-specified generic safety requirements. The difference with other similar proposals is that the generic safety requirements in the approach by Bitsch (2001) are expressed as complete requirements, therefore removing the need of combining the right pieces to build a requirement. His approach is intended to be used in two steps. In the first step a pattern is chosen from the catalog of available patterns

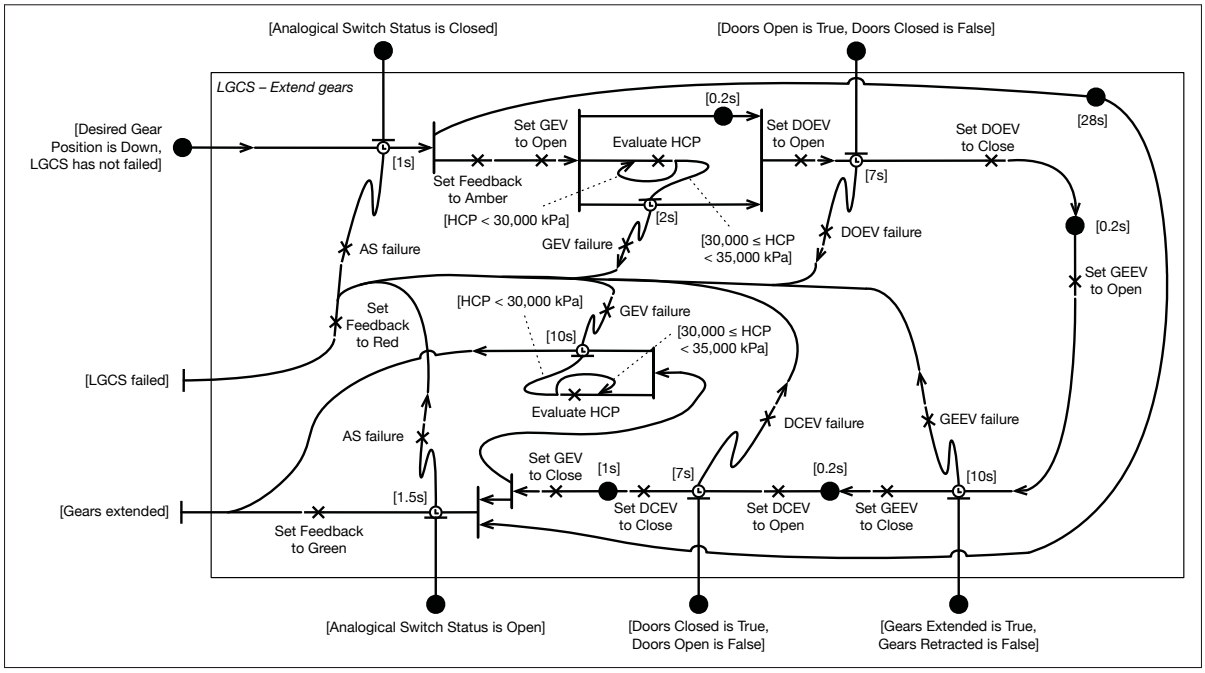


Figure 1.16 UCM behavioural model for a landing gear controller.

for safety requirements. The patterns are organized with a classification scheme that facilitates their identification. Each pattern is written as a formal formula (in various formalisms) and is explained in natural language so as to make it easier to understand and learn. In the second step, the selected pattern is applied to the corresponding system safety requirement. The result is a safety requirement that is expressed formally using a formal formula and also expressed with a correct formulation in natural language.

Bialy *et al.* (2015) propose the use of Horizontal Condition Tables (HCTs). An HCT defines a single function and, thus, describes a single behaviour that computes a single output. HCTs are a simplification of Stateflow truth tables. Figure 1.17 shows the structure of an HCT on the left and an example on the right. Each row of the table can be regarded as a decision, and the results column as the action section. There is no defined order in which rows are evaluated. For a table to properly (totally) define a function, two conditions must be satisfied: 1) rows' disjointness, and 2) completeness (*i.e.* all cases are covered).

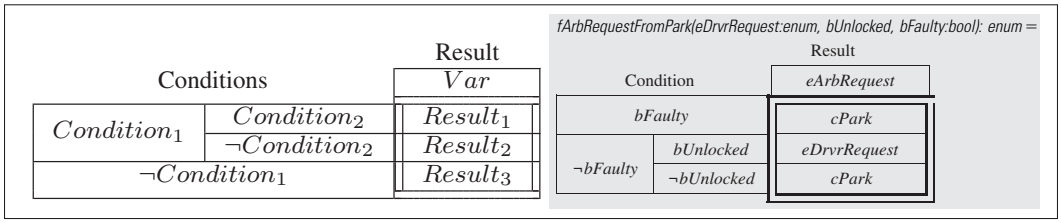


Figure 1.17 Structure of an HCT (left) and an example (right).
Extracted from Bialy *et al.* (2015) and Bialy *et al.* (2017), respectively.

Fifarek *et al.* (2017) developed the Specification and Analysis of Requirements (SpeAR) language and tool. Figure 1.18 presents an example specification with SpeAR. The language is designed to read like natural language. It has the formal semantics of Past Linear Temporal Logic (Past LTL), which supports proofs of critical properties about requirements using model checking (logical entailment and logical consistency). Logical entailment proves that the specified properties of the system are consequences of the assumptions and requirements. Logical consistency aims to identify conflicting assumptions and requirements. Other analyses and validations provided by the tool include type-checking. SpeAR can also capture natural language-based requirement statements but only for those requirements that cannot or are not intended to be formalized. SpeAR promotes the following structure for specifications: 1) inputs represent monitored data from the environment, outputs represent data sent to the environment, state represent the component's local data (not visible to the environment), 2) assumptions identify necessary constraints on the inputs, 3) requirements identify constraints that the component must guarantee in its implementation, and 4) properties represent constraints that the system should satisfy when operating in its intended environment,*i.e.* they validate that the requirements define the correct component behaviour and prove that certain undesirable conditions never arise.

Micouin (2008) proposes the theory of property-based requirements (PBRs) as a method for solving the problems (*e.g.*, ambiguity, inconsistency, incompleteness) of natural language-based requirements specifications. A PBR for a system Σ is defined as a constraint over a property P of an object O in Σ. The constraint enforces that the value of P is located within

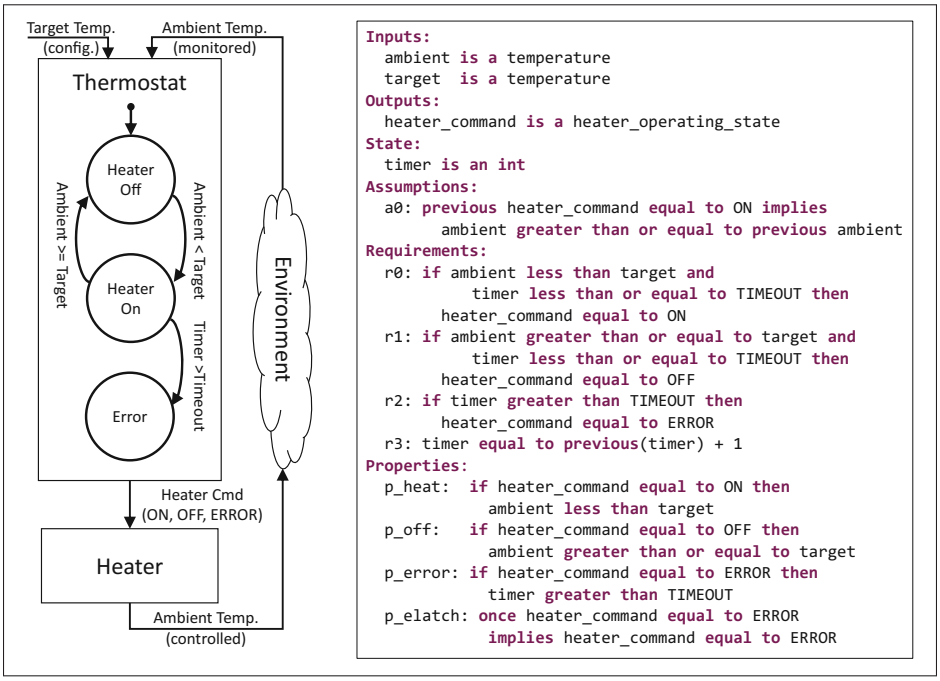


Figure 1.18 SpeAR specification for a thermostat. Extracted from Fifarek *et al.* (2017).

a domain D, which is a subset of $im(P)$ (*i.e.* the domain of possible values of P), when a condition C is met. Expression 1.1 formalizes the expression of a PBR. The expression reads as follows: *"[when condition C is met,] the value(s) of property P of object O shall be in the subset D of the set of possible values for P"*. The presence of a condition C is optional as indicated by the presence of square brackets. In the expression, the term Req is a mandatory, unique requirement identifier. The theory states that the conjunction of a finite set of PBRs $\{\text{Req}_n\}$ denotes the system Σ.

$$\text{Req: } [\text{when } C \rightarrow] \, val(O.P) \in D \subset im(P) \tag{1.1}$$

Gaucher & Génevaux (2017) present Argosim STIMULUS, a commercial approach to address the issue of early debugging and validation of requirements. Figure 1.19 illustrates a formalization for an HLR of a car's automatic light system. STIMULUS provides two key features: 1) expresses requirements and environment assumptions in a formal specification language yet

it is close to natural language, and 2) generates observers from such requirements that observe simulation results to identify requirement satisfaction under environmental assumptions.

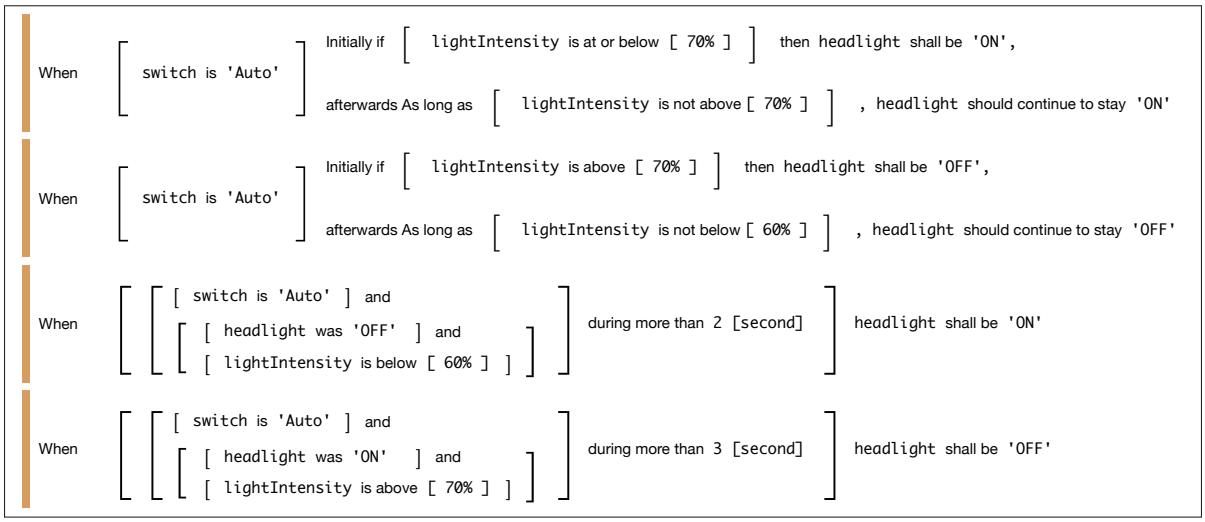


Figure 1.19 Formalization of an HLR for a car's automatic light system using STIMULUS. Adapted from Gaucher & Génevaux (2017).

The UML profile for Modeling and Analysis of Real-Time Embedded Systems (MARTE) (OMG, 2011) is a standardized graphical modelling language based on UML for describing real-time and embedded systems. MARTE adds to UML specific support for time and resource modelling (*i.e.* performance and schedulability) of both hardware and software. Software aspects can be allocated onto the hardware architecture. The modelling part of MARTE is intended for the specification of detailed system designs. The analysis part of MARTE concerns the provision of facilities to annotate the created models with quantitative information and other system properties to perform different analyses oriented towards system validation and optimization. The UML profile for MARTE is conceived as a hierarchy of subprofiles organized into four groups, as shown in Figure 1.20. The first group is the foundation for the other three groups and consists of subprofiles providing constructs for specifying non-functional properties, time representation (*e.g.*, chronometric, logical, synchronous), sets of resources and their usage, and the allocation (*e.g.*, time-related, space) of functionality to the entities responsible for their realization. The second group is dedicated to the model-based design of systems.

The third group is dedicated to the model-based analysis of systems. The fourth group holds annex subprofiles that enable additional facilities for modelling additional system aspects for contemporary computing platforms and structures.

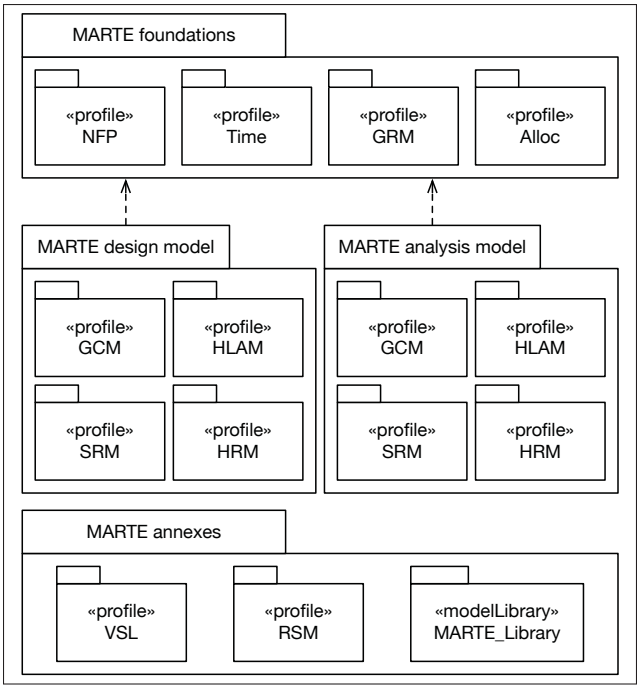


Figure 1.20 Architecture of the UML profile for MARTE. Adapted from OMG (2011).

1.3.3 Approaches handling design model heterogeneity

Safety-critical systems are complex systems combining physical and mechanical components, networking and software (Lee, 2010). Considerable amounts of effort throughout the years have evolved the engineering of these aspects along siloed trajectories. As Sztipanovits (2007) remarks, this does not translate into an increased design complexity but, instead, the opposite. Separately describing orthogonal system characteristics is the main strategy for handling and reducing complexity. However, when parallel development teams are designing such systems, integration issues may arise leading to inconsistent constructions prone to experience unintended or emergent behaviour. Thus, ensuring consistency between heterogeneous design models is an important problem when handling heterogeneity. Several approaches exist that

present structured ways of handling model heterogeneity. The following subsections examine three lines of work in this regard: 1) model composition, where approaches compose a new model from heterogeneous models, 2) model integration, where approaches bring all models into participation through an additional model or leveraging one to include references to the others, and 3) consistency management.

1.3.3.1 Model composition

Eker *et al.* (2003) propose an approach to compose different, heterogeneous models for the purposes of design analysis and code generation. The approach follows a hierarchical strategy to structure a complex model as a tree of smaller, lower-level model compositions. The aggregated compositions form a network of interacting components. Each of these components is semantically significant in the sense that to the next higher level of the hierarchy they are considered to be atomic. Atomicity is achieved by specifying both flow of data and flow of control, which effectively defines how a computation is going to take place within the structured components. Yu *et al.* (2011) propose an approach for composing and integrating heterogeneous models. In their approach, the system's functional behaviour is modelled with synchronous data flow and state machine diagrams. For its part, the system's architecture is represented with AADL as an assembly of components. These heterogeneous models are automatically transformed into a common model that allows for an integrated interpretation and simulation. The common model is expressed as processes with the SIGNAL language. Yu *et al.* (2011) also describe how dynamic analyses can be performed during simulated system executions.

Bozga *et al.* (2004) propose an environment for modelling and validating heterogeneous real-time systems. The environment comprises 1) an intermediate language called the IF language, 2) a toolset for transforming input models (described in, *e.g.*, UML) into the IF language and calculating possible executions to generate test cases, and 3) a methodology for using the toolset. OpenDo (2011) proposed the Project P framework to 1) verify the semantic consistency of systems described using safe subsets of heterogeneous modelling languages (*e.g.*, Simulink, SysML, MARTE, UML), ranging from behavioural to structural languages and pre-

senting a synchronous and asynchronous semantic, and 2) generate optimized source code for multiple programming and synthesis languages. Despite all these interesting features, tools and documentation other than what has been presented here for the IF language and the Project P framework are not publicly available.

1.3.3.2 Model integration

Various studies present integrative approaches for safety-critical system design with mixes of different modelling languages. Commonplace modelling languages like UML, SysML, Simulink and Stateflow, gather the most attention in this research area. Farkas *et al.* (2009) present an heterogeneous design modelling approach integrating UML and Simulink design models. They built the approach in an industrial context of automotive software engineering. The approach handles two aspects. On the one hand, the modelling of embedded software using different modelling languages. On the other hand, the integration of legacy artifacts into the design models. The work relies on UML to capture a high-level view of the system and to act as a master model. A custom-built UML profile supports the latter. The profile's stereotypes help collect references to all other models and legacy artifacts defining specific system aspects.

Sakairi *et al.* (2012) propose an heterogeneous design modelling approach integrating SysML and Simulink design models to achieve system verification through simulation in MATLAB. The approach starts with the system design using the different SysML diagrams and then automatically generate Simulink models from the information captured with SysML. A custom-built SysML profile supports the generation process and collects references to specific system aspects for which Simulink is a more appropriate modelling language. Similarly, Bombino & Scandurra (2013) present an heterogeneous design modelling approach integrating SysML and Simulink design models with the goal of providing simulation. They propose a co-simulation framework to balance the strengths of both languages. The approach also starts with the high-level design of the entire system using the different SysML diagrams. The difference is that discrete-time behaviour is further designed with SysML while continuous-time behaviour is further designed with Simulink. Additionally, their technique for co-simulation maps the dif-

ferent models at the code level. Thus, code is generated from all the models along with glue code representing the model interactions.

Ferrari *et al.* (2013) present the model-based development of the Automatic Train Protection (ATP) System of the Metrô Rio, in the city of Rio de Janeiro, Brazil (see Figure 1.21). In particular the system's design involved the combination of UML, Simulink and Stateflow. Starting from the system requirements, they developed the system architecture in the form of a UML component diagram. They then manually translated this UML architecture into Simulink subsystem blocks, each with its behaviour specified in terms of a set of Stateflow charts. The UML and Simulink models exhibit a complete overlap. Their work in the UML-to-Stateflow translation led them to the definition of modelling rules extending the MAAB guidelines in order to constrain the Simulink modelling language in parts where they considered it is semantically ambiguous. No details are given on the mappings between the models, or on how the UML components' provided and required interfaces are mapped onto the Simulink subsystem blocks' inputs and outputs.

Sjöstedt *et al.* (2008) define a procedure for transforming Simulink models into UML composite structure and activity diagrams in order to facilitate software design. As part of their work, they investigated structural and behavioural mappings between subsets of these two modelling languages. Simulink (subsystem and basic) blocks are mapped to UML classes, and Simulink in and out ports are mapped to UML ports. The behaviour represented by the interconnection of blocks in the Simulink model is mapped to UML activities. A custom-built UML profile captures the specific behaviours of the Simulink basic blocks. When a Simulink basic block is transformed to UML, the corresponding stereotype is applied to the resulting class. Likewise, Tanaka *et al.* (2017) propose a model transformation tool to transform Simulink models into UML composite structure and interaction diagrams (see Figure 1.22). The difference is that Simulink in and out ports are mapped to UML classes. These classes have accessor and/or mutator methods depending on if they represent an in or an out port. A controller class calls the mutator methods to produce the outputs. However, when designing safety-critical software, UML input and output parameters in operations are preferable to enable static analyses.

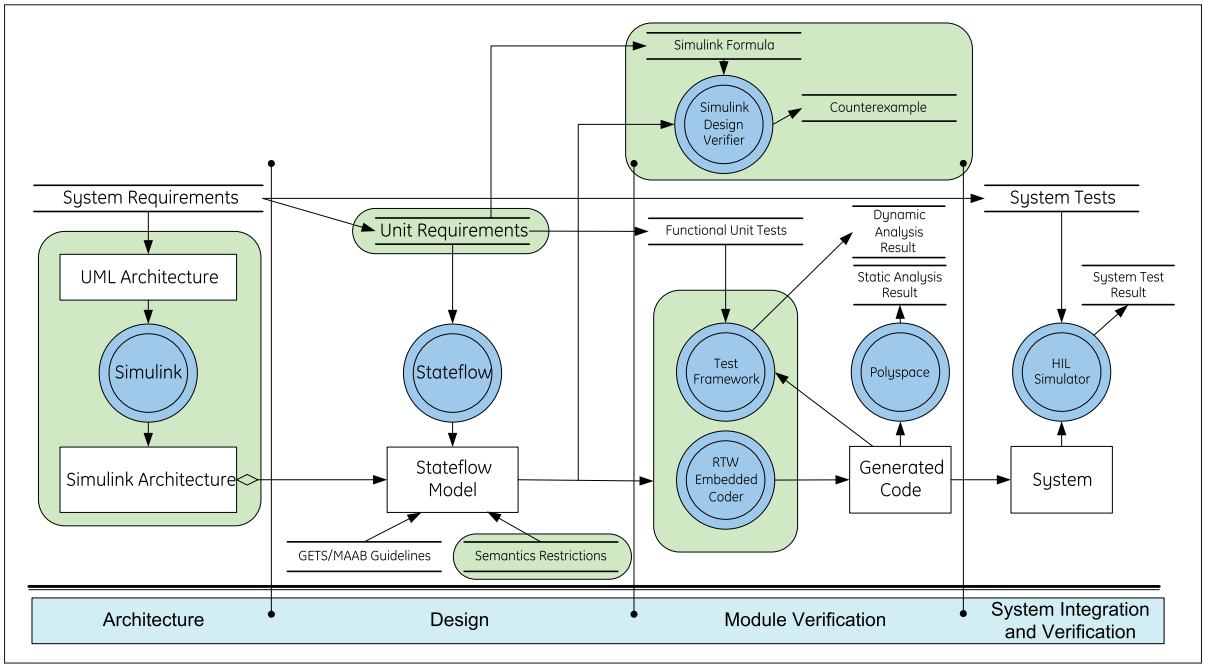


Figure 1.21 Development process for the ATP system in the Metrô Rio. Extracted from Ferrari *et al.* (2013).

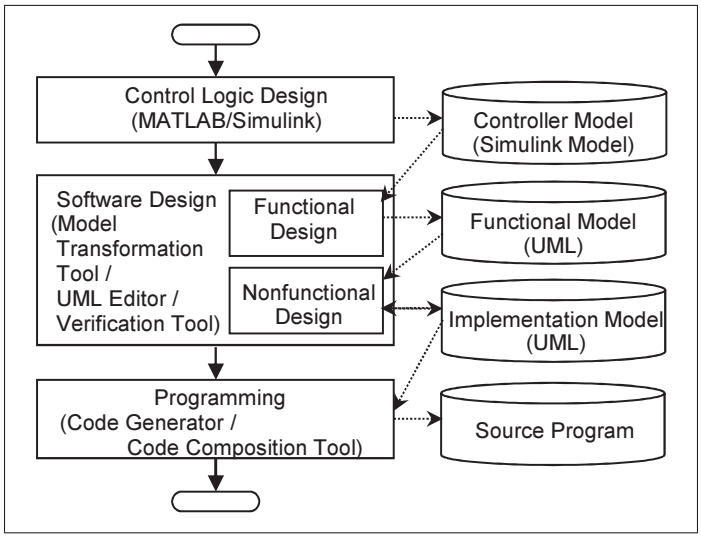


Figure 1.22 Proposed development flow by Tanaka *et al.* (2017). Extracted from Kuroki *et al.* (2016).

1.3.3.3 Consistency management

A number of approaches addressing the problem of consistency in heterogeneous and multi-viewpoint design models have been proposed. The ARCADIA/Capella (Roques, 2016) MDE solution is a successful approach deployed in several industry companies. The approach provides methodological guidance through successive engineering phases from operational and system need analyses to system design (see Figure 1.23). It prescribes three interrelated activities: 1) need analysis and modelling, 2) architecture building and validation, and 3) requirements engineering. Four features can be highlighted from the methodological guidance: 1) engineering-wide collaboration through a shared reference architecture, 2) architectural complexity management, 3) trade-off analysis for definition of optimal architectures, and 4) information refinement with traceability between the different engineering levels. In order for ARCADIA/-Capella to support all these features, it proposes its own DSML based on SysML.

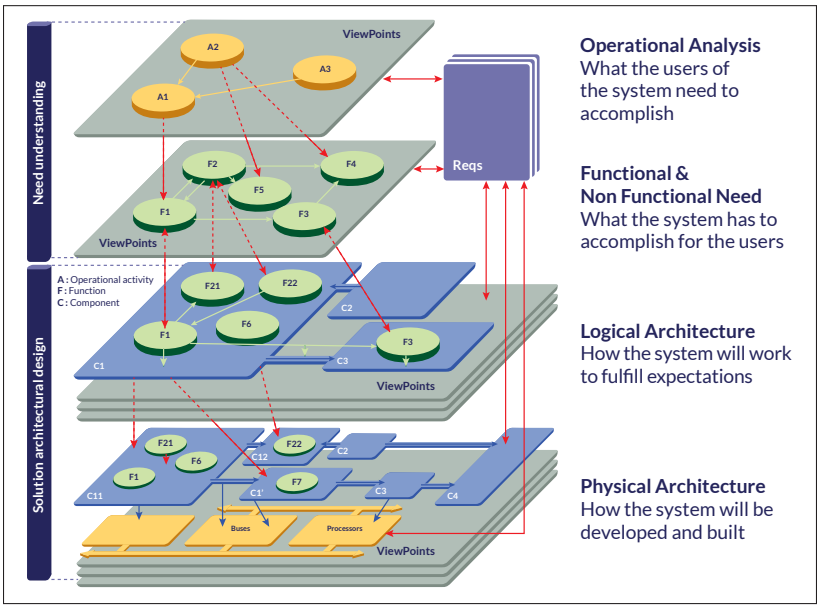


Figure 1.23 The ARCADIA engineering approach. Extracted from Roques (2016).

Finkelstein *et al.* (1994) present an approach for specifying logical rules on how to handle inconsistencies in multi-viewpoint design models. The approach is based on temporal logic.

Logic rules are expressed in the context of potential inconsistencies that can occur in the models. Almeida da Silva *et al.* (2010) provide automated methodological support for consistency management in heterogeneous design models. The approach ensures the software process is being performed as specified and that process guidelines are not being violated. In order to process design models, these are forced to be represented as the sequences of actions needed for their construction. Like with the previous approach, logic rules defined over the model construction action sequences identify undesired action patterns that (probably) cause inconsistencies between the design models. Their implementation monitors the design model's construction actions to ensure they do not have a negative impact on the other design models.

Dijkman *et al.* (2008) present an approach with re-usable consistency rules for preserving consistency in multi-viewpoint design of enterprise information systems. The approach relies on the abstraction of concepts from the different modelling languages to build a set of basic concepts over which consistency rules can be defined. El Hamlaoui *et al.* (2018) propose an approach aimed at maintaining consistency when design models evolve. The approach consists of two steps: matching and consistency management. The first step analyzes input design models to identify correspondences between them. A model of correspondences is created storing the identified correspondences. The second step defines an observer over every correspondence. The observers detect changes done to the original model elements. Changes are then classified, scheduled and prioritized for processing.

1.4 Discussion

In this section we discuss and elaborate on what is observed from our literature review presented in Section 1.3. We also identify what are the considerations for reusing or borrowing relevant features from the previous approaches and the opportunities for filling gaps in the context of developing and providing a comprehensive model-based approach that is compliant with DO-178C.

1.4.1 Approaches supporting certification

An important observation is that DO-178C is represented in natural language, which brings along several drawbacks: it is open for interpretation and contains inconsistencies and contradictions in its text. As a consequence, its understanding and the clear determination and communication of the evidence to collect is hindered. Thus, the modelling of DO-178C is required in order to lead to any form of automation for certification compliance. Although DO-178C has been modelled in the literature (*e.g.*, Ruiz *et al.* (2016); de la Vara *et al.* (2016); Zoughbi *et al.* (2011)), such models are only partially available, which limits their reuse. Furthermore, DO-178C belongs to the category of prescriptive safety standards (Hawkins *et al.*, 2013; Ruiz *et al.*, 2016). This means that satisfying its compliance needs is tightly linked to following the activities and generating the evidence it requires for the corresponding software level assigned to the system that is going to be built. The reviewed approaches for modelling compliance needs (*e.g.*, Ruiz *et al.* (2016); de la Vara *et al.* (2016); Zoughbi *et al.* (2011)) lack the possibility of parameterizing its constructs to the assigned DO-178C software level for the software to be built, which is a feature that can help facilitate compliance efforts. A first step was then to work on the modelling of DO-178C. This was developed with the help of a masters student and the results are reported in his masters thesis (Metayer, 2018). The modelling of DO-178C was also presented at the 9th International IEEE Workshop on Software Certification (WoSoCer 2019) hosted at the 30th International Symposium on Software Reliability Engineering (ISSRE 2019) (Metayer *et al.*, 2019).

Hawkins *et al.* (2013) have made a comparative examination that contributes to the discussion of the use of assurance cases for safety assurance in the development of DO-178C-compliant safety-critical avionics software. However, an approach on how to bring the two together is required. The SACM metamodel OMG (2018) can be used to record the assurance arguments, but some shortcomings need to be addressed such as the lack of means to bridge its argumentation concepts to DO-178C safety compliance needs. Stallbaum & Rzepka (2010) is able to bridge test information with compliance needs and assurance, and from which test cases can be generated. These are relevant features that can be borrowed.

A remark to make about traceability is that even though having an active traceability support is a requirement of most—if not all—software safety-related regulations in order to ensure an adequate safety-critical software development, the reviewed approaches do not generally include or highlight elements inside or outside (*e.g.*, tools) their proposal for this purpose. We note, however, no variability in terms of how traceability links (or traces) are established in the approaches. We believe that since all the reviewed approaches are in the context of MDE, they are somewhat constrained in the definition of traces and traceability schemes. All the reviewed approaches that support traceability limit it to the creation of links among the modeled elements mostly through explicit (direct) and implicit (transitive) relationships. Traceability, and specifically traceability in the context of MDE, is an active research field of software engineering.

The model-based toolchain described by Eisemann (2016), composed of commercially available tools, seems to be comprehensive. Nonetheless, toolchains of sorts suffer from vendor lock-in and usually target specific modelling languages. As is the case with such a toolchain, specification models are limited to be just Simulink models. Modelling constructs provided by Simulink are restricted to be from electrical and electronic domains, which are not suitable in all cases for modelling all system aspects (*e.g.*, software). Simulink models are also limited to block and state machine notations, which are not appropriate to model all requirements.

1.4.2 Approaches supporting requirements specification in the context of certifiable safety-critical systems development

In all of the DO-178C development life cycle, the requirements and verification phases are two of the foremost development activities. As Feiler (2010) points out, many of the errors found in safety-critical systems have their origin in the requirements phase. Requirements are of utmost importance since they must convey a clear understanding of the problem that will be solved. The DO-178C certification process acknowledges this and centres its most stringent compliance needs around 1) proper argumentation and traceability of requirements to design and certification concepts, and 2) requirements-based verification. However, current industry practices

have requirements specified using constrained natural language and managed with the help of textual requirements databases, like IBM DOORS (Potter, 2012; Blouin, 2013). Although the use of a constrained natural language brings some discipline and rigour to requirement specifications, this practice is still focused on writing requirements in human-readable prose. This inevitably raises problems for satisfying certification needs. Natural language indeed facilitates communication between stakeholders but it is not a suitable form of specification for supporting interrelationships, decomposition, requirements-based analyses and testing (Moy *et al.*, 2013).

Tables 1.1, 1.2 and 1.3 provide a summary of the reviewed approaches in this regard. Several requirements modelling languages exist that are based on UML. SysML (OMG, 2017a) adds on top of UML specific constructs for capturing requirements and their interrelationships. Nonetheless, requirements are expressed with simple natural language statements and the semantics of relationships are ambiguous and open to interpretation. MARTE (OMG, 2011) is intended for describing real-time and embedded systems. However, MARTE is focused on non-functional aspects of requirements. Thus, it lacks the necessary constructs to create a complete requirements specification on its own. SafeML (Zoughbi *et al.*, 2011) extends UML for capturing safety-related requirements allocated to software and enabling the monitoring of the design and implementation of the software with regard to the provided safety requirements. However, it captures requirements in natural language, meaning the safety-related information must be manually extracted to support their verification. The avionics industry considers that UML, with profiles such as SysML and MARTE, provide better long term sustainability and interoperability over completely custom-built domain-specific languages (Le Sergent *et al.*, 2016). Zoughbi *et al.* (2011) list the benefits justifying the use of UML for safety-critical avionics software.

Leveson *et al.* (1994), Micouin (2008), Amyot (2003), Blouin (2013), Bialy *et al.* (2015), Fifarek *et al.* (2017) allow the expression of semantically richer requirements than natural language statements. While RSML (Leveson *et al.*, 1994) has been successfully used in the avionics domain for the specification of the TCAS II, a complete formalization of the language

Table 1.1 Summary of model-based approaches supporting requirements specification

Approach	**Goal**	**Specification technique**	**Scope**
SysML	Capture requirements and their interrelationships	Natural language	Requirement statements, and their relationships to other requirements and other elements (*e.g.*, design elements, test cases)
SafeML Zoughbi *et al.* (2011)	Capture safety-related requirements	Natural language	Requirement statements, requirement refinement, and relationships to design elements
Nejati *et al.* (2012)	Traceability management	Natural language	Requirement statement and relationships to design elements
Biggs *et al.* (2016)	Model safety-related concerns	Natural language	Requirement statements and metadata
RDAL Blouin (2013)	Requirements definition and analysis	Natural language	Requirement statements, and relationships to design elements and verification activities
ReqSpec Feiler *et al.* (2016)	Requirements specification language for AADL	Natural language with dynamic evaluation of property values	Requirement statements, relationships to design elements, and formalized declarations of requirement statements

was never created and the language is not openly available (Whalen, 2000). Furthermore, the use of hierarchical abstraction obscures the separation between HLRs and LLRs. The language even allows users to include design-level information in the requirements specification, a practice that is greatly discouraged by DO-178C.

Table 1.2 Summary of model-based approaches supporting requirements specification (Continued)

Approach	**Goal**	**Specification technique**	**Scope**
UML	Create views of requirements	Behaviour models	Expected behaviour and modes of operation
RSML Leveson *et al.* (1994)	Model black-box behaviour and assumptions on external components	Function mapping inputs and outputs in the face of disturbances	Inputs, outputs, expected behaviour and modes of operation
UCM Amyot (2003)	Discover, specify and review requirements	Behaviour models	Expected behaviour

Recall that with RDAL (Blouin, 2013) requirements can be expressed using both natural language statements and by referencing UCMs (Amyot, 2003). Formalizations of the require-

ments may also be referenced from their expressions using constraint languages (*e.g.*, OCL). A major drawback of this is that neither RDAL nor UCM were designed to have explicit traceability to certification concepts. Moreover, UCM, like RSML, is prone to convey design information.

Although SpeAR has been developed with certification compliance in mind, it has a very generic vocabulary that makes it difficult to argument certification compliance. Furthermore, it does not enforce certain mandatory information (*e.g.*, rationale). HCTs, PBRs and SpeAR force requirements to be captured in an already structured form, which restrains their adoption in industry. Moreover, they lack constructs for expressing timing constraints and clocks, interrelationships between requirements, and the identification of derived requirements.

Table 1.3 Summary of model-based approaches supporting requirements specification (Continued)

Approach	**Goal**	**Specification technique**	**Scope**
Bitsch (2001)	Simplify specification of safety requirements	Pattern-based catalog of complete, generic safety requirements written as formal formulas and in natural language	Requirement statements
HCTs Bialy *et al.* (2015)	Formalize specification of functional requirements	Function computing a single output from given inputs	Inputs, output, and mapping function
SpeAR Fifarek *et al.* (2017)	Specification and analysis of requirements	Formal syntax based on Past LTL that reads like natural language	Requirement statements
PBRs Micouin (2008)	Solve the problems of natural language-based requirements	Constraints enforcing a property whenever a condition is met	Requirement statements
STIMULUS Gaucher & Génevaux (2017)	Address the issues of early debugging and validation of requirements	Formal syntax that reads like natural language	Requirement statements
MARTE	Modelling and analysis of non-functional requirements	Semi-formal syntax	Timing constraints, resources (hardware and software), and allocation of software to hardware

1.4.3 Approaches handling design model heterogeneity

As argued in the previous section, effective system modelling requires a heterogeneity of modelling mechanisms focused around specific system aspects (Rozanski & Woods, 2011; Lee, 2010). Regarding model heterogeneity, the previous section identified some relevant approaches that address model heterogeneity. Despite their offerings, however, there are some limitations for their use. The approaches proposing model composition (Eker *et al.*, 2003; Bozga *et al.*, 2004; OpenDo, 2011) do not target certification compliance needs that have to be taken into account when dealing with model heterogeneity. Furthermore, they do not support the modelling of evidence for certification. The approaches presenting integrative approaches for heterogeneous design with UML/SysML, Simulink and Stateflow do not cover the entire extent of industrial design scenarios. Actually, they are representing different combinations for what are mixed use cases of these languages. Moreover, they can provide knowledge to be used as part of heterogeneous design guidelines. In addition to this, is the fact that they do not provide insight on how consistency is ensured when overlapping elements exist in the models. Some of these approaches, like the ones by Sakairi *et al.* (2012), Sjöstedt *et al.* (2008) and Tanaka *et al.* (2017), use model transformations in order to ease the integration of the modelling languages. Model transformations can be regarded as a mechanism to ensure consistency between the two models. However, their use has a drawback when it comes to independent model manipulation, which is desired when developing heterogeneous systems. Manual work may be required to verify and fix the consistency between models if bidirectional transformations are not provided.

Despite ARCADIA/Capella's offerings for consistency management, its major drawbacks are that 1) it focuses on presenting different system model views created by different engineers using only its proposed DSML, and 2) it does not verify conformance to design standards. A consistency management approach could, in fact, be used to complement ARCADIA/Capella 1) when multiple modelling languages are employed for system design, and 2) for providing verification of conformance to design standards.

The major drawback of consistency management approaches like the ones from Finkelstein *et al.* (1994) and Almeida da Silva *et al.* (2010) is that model construction actions and logic rules are cumbersome to use for the expression of model mapping rules and design guidelines. Furthermore, the need for a formal background to specify logic rules is a barrier for adoption in industry. Furthermore, these approaches ensure consistency by construction and, thus, do not explicitly store mappings between the different design models. Storing mappings is beneficial for supporting verification of design process outputs as well as for supporting further development activities. Dijkman *et al.* (2008) present an approach focused on enterprise information systems, which have different characteristics than safety-critical systems. Regardless, the major drawback of this approach is the abstraction of concepts from the different modelling languages to build a set of basic concepts over which consistency rules can be defined. Such abstractions are not always possible, as is the case when a modelling language lacks an explicit construct that is present in another one. The major drawback of the approach by El Hamlaoui *et al.* (2018) is that it is directed at maintaining consistency only when the design models evolve, *i.e.* after the design models are created, correspondences have been identified and a corresponding observer is attached to each correspondence. Thus, the approach fails to manage consistency while the design models are being created independently, moment at which changes could be introduced that prevent the identification of correspondences.

1.5 Chapter Summary

This chapter first presented research background relevant to the scope of this thesis. DO-178C is a conceptual guideline identifying the set of *best practices* to take into consideration during the development of software for airborne systems and equipment. These *best practices* are stated in the form of *objectives*, which have to be achieved by carrying out a set of explicitly defined *activities*. Activities produce evidence (*e.g.*, plans, requirements, design description), known as *data items*, required for certification. DO-178C addresses, in separate supplement documents, new considerations and practices in two key areas of contemporary software development: 1) model-based development and verification (DO-331), and 2) object-oriented

technologies and related techniques (DO-332). MDE conceives the system development life cycle as a process of creation, iterative refinement and integration of models. In order to do this, MDE makes use of DSMLs to provide users with a working environment where they can directly manipulate domain concepts. The chapter synthesized how modelling is done in the set of modelling languages used by the industry partners. It then presented a systematic methodology to build custom DSMLs for regulation certification.

The remainder of the chapter presented the literature review of this thesis. Existing approaches can be divided into three lines of work: 1) approaches supporting certification, 2) approaches supporting requirements specifications in the context of certifiable safety-critical systems development, and 3) approaches handling design model heterogeneity. Several requirements specification languages exist that have been used in the context of certifiable safety-critical software development. Some of them only support the specification of natural language-based requirements. Others allow the expression of semantically richer requirement statements than natural language statements. However, all of the latter force requirements to be captured in an already structured form, which restrains their adoption even when they can enable requirements-based analyses and testing. Moreover, none of these languages provides sufficient support to help meet certification compliance needs. A number of existing studies have addressed mapping relationships between heterogeneous modelling languages. However, they target pairs of specific modelling languages under particular design scenarios without devising a general approach. Other existing works present consistency management approaches for heterogeneous design models but do not explicitly provide the syntactical and semantical relationships between them. Furthermore, none of them tackled the issue of verifying conformance to design standards.

CHAPTER 2

THE LANDING GEAR CONTROL SOFTWARE

There has been increasing work around more up-to-date, effective software engineering technologies to aid avionics software providers in reducing software and development complexities and to support them in their DO-178C certification endeavours (Pettit *et al.*, 2014). However, the legal and safety implications that public scrutiny may bring onto avionics industry manufacturers make them keen on keeping their projects confidential. This situation hinders research and education on the engineering and certification of avionics software. It is, therefore, necessary to have detailed and open documentation of software for airborne systems that 1) is readily available for research and education, and 2) can serve as a benchmark for supporting the evaluation of different (existing and new) model-based approaches.

A number of studies have supported their work with examples and case studies of avionics systems that involve some piece of software (*e.g.*, Leveson *et al.* (1994); Zoughbi *et al.* (2011); Wu *et al.* (2015); White & Reza (2012)). Others report on the development of avionics systems that rely on software for controlling the behaviour of such systems (*e.g.*, Schamai *et al.* (2015); Boniol & Wiels (2014)). Apart from most of them not being openly available, the case studies, as described in the previous works, do not allow for their reuse as comprehensive references for avionics software development in conformance with DO-178C. Boniol & Wiels (2014), in particular, introduced a case study of a landing gear system. Their descriptions provide a detailed view of its structure and inner workings, specially of the hardware elements. Software requirements are given but are not developed in accordance with DO-178C. Thus, building on such descriptions, we developed an avionics software case study for a landing gear control software (LGCS) in compliance with the DO-178C guideline, and the DO-331 and DO-332 supplements.

The LGCS will be used as a running example in the rest of this thesis to illustrate our proposed approach. In this chapter, the LGCS illustrates the motivations that led us to 1) define a systematic and automated method for assisting engineering teams in ensuring consistency of het-

erogeneous design models of safety-critical systems, and 2) develop a requirements modelling language that provides a requirements specification infrastructure for software development and certification according with DO-178C. It is to be noted that the descriptions given in this chapter are simplified to provide better focus for the context of this thesis. Appendix II contains the requirements specification and design in textual natural language statements. The complete version of the specification and design is available online (Paz & El Boussaidi, 2017). The documentation was built iteratively and verified by domain experts from the industry partners. It is organized in chapters, each corresponding to one of the DO-178C data items output by the planning, software requirements and software design processes. It took three iterations to obtain a favorable perception from the practitioners involved.

Section 2.1 first summarizes related work to motivate the need for the detailed documentation of an safety-critical avionics software. The next three sections synthesize several DO-178C data items generated for the LGCS. Section 2.2 presents the Plan for Software Aspects of Certification (PSAC), going through an overview of the system, its certification considerations, and development methodology. Section 2.3 presents the Software Requirements Data with the development of the system requirements allocated to software. Section 2.4 presents the Design Description, which includes the architecture and specification of low-level requirements. Finally, Section 2.5 discusses the challenges and issues of building such specification and design.

2.1 Related Work

Existing related work can be divided into 1) generic work in the field of safety-critical software requirements engineering and design that support their findings and evaluations with an example or case study of an avionics system, and 2) existing reports on the development of a software-intensive avionics system.

A number of studies have supported their work with examples and case studies of avionics systems that involve some piece of software. Leveson *et al.* (1994), the proposers of RSML, apply their proposed language on an aircraft Traffic Collision Avoidance System level II (TCAS

II). The resulting requirements model was adopted by the US Federal Aviation Administration (FAA) as the official requirements specification for TCAS II. The specification, thus, had to be developed with industrial-grade quality and in accordance with the applicable regulation at the time (*i.e.* DO-178B). However, the TCAS II specification is not openly available for research. Furthermore, the authors do not expand on the aspects of compliance with regulation.

Berkenkötter & Hannemann (2006), Zoughbi *et al.* (2011) and Wu *et al.* (2015) illustrate the capabilities and usage of their proposed UML profiles through case studies of tramway network and avionics systems. None of these systems is openly available for research. Additionally, the descriptions given for the case studies are missing important details of system requirements. Also, the provided system designs lack detailed behavioural specifications.

The work by White & Reza (2012) falls into the first category of related work, although, it has some elements of the second category as well. They validate their approach on an example requirements specification for the display of the fuel quantity in an aircraft. Their HLRs and LLRs specifications, however, have some shortcomings as some of the LLRs presented are, in fact, HLRs. Presenting them as LLRs makes the specification of the HLRs go against the characteristics of good requirements specification since they are incomplete and, thus, the presented LLRs could not have been developed from them. On the other hand, the authors report on some of the difficulties that arise when developing a requirements specification for an avionics software under DO-178C. The difficulties they describe are focused on the capturing of derived requirements, a special type of requirement under DO-178C that is characterized by not being directly traceable to higher-level requirements because it specifies behaviour beyond what is specified in the higher-level requirements.

Only a small number of reports on the development of software-intensive avionics systems were found in the literature. Ferrari *et al.* (2013) present the model-based development of the Automatic Train Protection (ATP) System of the Metrô Rio, in the city of Rio de Janeiro, Brazil. They present an overview of the system and the process followed to develop it. How-

ever, most of the details are on code verification and formal verification. The authors do not mention specific aspects of their compliance with regulation.

Schamai *et al.* (2015) develop a requirements specification for the design of the secondary flight system of an aircraft. Particularly, the authors describe the spoiler activation requirements. However, the specification is not openly available for research either. Moreover, although the system heavily relies on software for its operation, as it is intended for a fly-by-wire aircraft, the requirements that are presented are not developed taking into account the DO-178C guideline. Boniol & Wiels (2014), on their part, introduced the landing gear system (LGS). Their descriptions provide a detailed view of a conventional tricycle landing gear system, and the detailed structure and inner workings of the elements involved, especially the hardware elements. However, despite being both complex and representative of industry needs, the software requirements given are not developed following the DO-178C guideline and, thus, fail to present the detailed reference expected in software development for airborne systems. Moreover, they fail to give values of key system properties that should be part of system requirements. Nonetheless, these descriptions offer a good starting point to build upon them. Thus, they have been taken to develop the detailed reference for the LGCS synthesized in this chapter.

2.2 Plan for Software Aspects of Certification (PSAC)

2.2.1 System overview

The overall system is a landing gear system (LGS) whose undercarriage is retractable and arranged in a conventional tricycle configuration as illustrated in Figure 2.1. The undercarriage is composed of two main wheel assemblies situated one under each wing of the aircraft and a nose wheel assembly at the front of the aircraft. The wheel assemblies, or gears, are retracted into compartments inside the wings and fuselage of the aircraft and concealed behind doors after the aircraft has taken off and began the climb phase. Conversely, at the beginning of the landing phase the gears must be extended for the aircraft to land. The doors covering the

wheel compartments need to be open prior to and closed after any retraction or extension is performed.

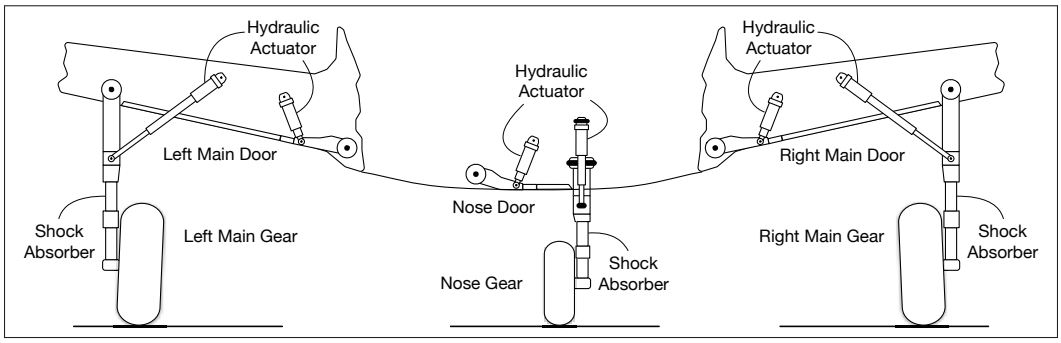


Figure 2.1 Illustration of an aircraft's landing gear system.
Adapted from Paz & El Boussaidi (2017).

The behaviour of the landing gear system is managed by the landing gear control software (LGCS) that runs in a digital controller. Figure 2.2 shows an overview of the LGS and the operational context of its constituent entities. The aircraft's cockpit accommodates the system's pilot interface. From it, the pilots provide the desired gear position to the system through a gear lever. A color-coded light set indicator provides them with information about the state of the gears and the system itself. All the gears and doors are moved through the sequential actuation (*i.e.* opening and closing) of five hydraulic electro-valves (EV) that control the oil pressure in the hydraulic circuit to which the gears' and doors' hydraulic actuators are connected. One general EV pressurizes the hydraulic circuit and the remaining four specific EVs (*i.e.* a door opening, a door closing, a gear retraction and a gear extension EVs) move their attached mechanical part. Seventeen sensors are in charge of monitoring the state of the various mechanical parts (*i.e.* one analogical switch sensor, one hydraulic circuit pressure sensor, three door open sensors, three door closed sensors, three gear retracted sensors, three gear extended sensors and three gear shock absorber sensors). For redundancy purposes, all sensors perform three independent readings describing simultaneously the same situation. At least two of the three readings must have an equal value for the LGCS to trust the sensor. To prevent any motion of the gears without the explicit order of the pilot or copilot, the LGS has an analogical switch to enable and disable the actuation of the general EV. The analogical switch is mechanically

closed each time there is a change in the gear lever's position. Thus, successful actuation of the general EV is only possible when the analogical switch is closed.

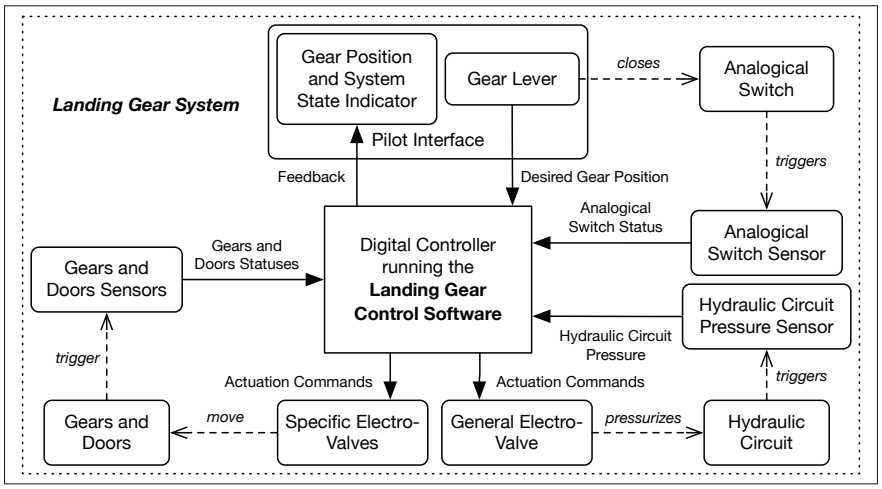


Figure 2.2 System overview. Extracted from Paz & El Boussaidi (2018).

The LGCS focuses on two primary needs: 1) to sequence the commands that actuate the EVs for the movement of the gears and doors to the given desired gear position taking into account the inputs from the different sensors, and 2) to display visible cues indicating the state of the gears and the system through the indicator in the pilot interface. In general, the LGCS controls the EVs at the beginning of the climb phase and at the beginning of the landing phase, when the pilots move the gear lever to the *Up* and *Down* positions, respectively. The LGCS is idle until a change in the gear lever position. When the gear lever is switched from the *Down* position to the *Up* position after the aircraft has taken off and has begun the climb phase, the LGCS begins the retraction sequence. The LGCS displays the amber color in the gear position indicator and opens the general EV to pressurize the hydraulic circuit. Once the hydraulic circuit is pressurized, the door opening EV is opened. After all the doors are open, the LGCS closes the door opening EV. The LGCS evaluates if the gear shock absorbers are relaxed or not. If the shock absorbers are relaxed, the LGCS opens the gear retraction EV. If the shock absorbers are not relaxed, the doors are closed and the sequence is aborted. Once all the gears are retracted, the LGCS closes the gear retraction EV and opens the door closing EV. After all the doors

are closed, the door closing EV is closed and the general EV is closed afterwards. The LGCS turns off the gear position indicator. When the aircraft begins the landing phase and a change in the gear lever position is sensed, the LGCS performs the extension sequence. The extension sequence follows the same steps as the retraction sequence except for the evaluation of the shock absorbers and the LGCS actuates the gear extension EV instead of the gear retraction EV. When the gears are all extended, the LGCS displays the green color in the gear position indicator. Figure 2.3 shows the three main states of a wheel assembly.

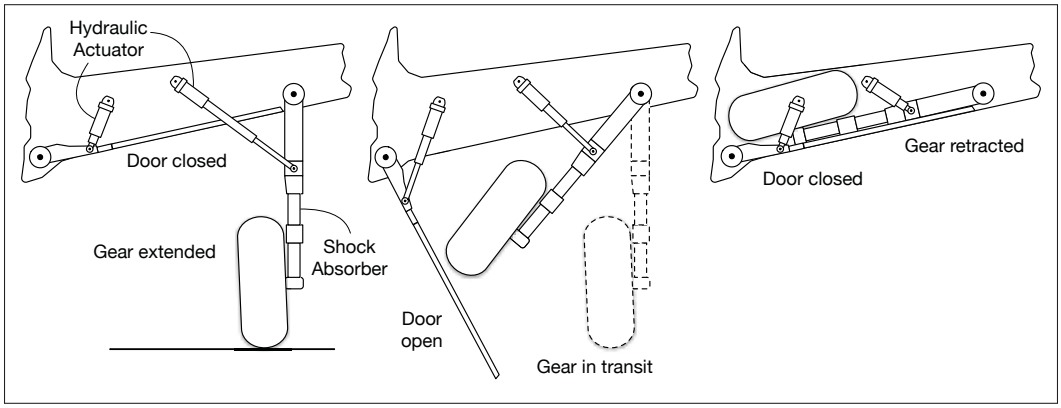


Figure 2.3 Illustration of the main states of a wheel assembly (from left to right): gear extended, gear in transit and gear retracted. Adapted from Paz & El Boussaidi (2017).

2.2.2 Certification considerations

The LGCS performs a critical function for the aircraft, however, in the event of a failure condition the pilots can manually crank the gears into their extended positions. Thus, it is assumed that the software could cause or contribute to a major failure condition, meaning it would significantly reduce safety margins and create a significant increase in crew workload as the LGS would need to be manually actuated. Additionally, it could cause discomfort to the passengers, flight crew and cabin crew due to the emergency landing that needs to be attempted. Failure of the LGCS may also lead to injuries to any person on board the aircraft. Consequently, the LGCS is considered to be a *Level C* software.

The following relevant System Hazards (SHs) were identified during the safety assessment process involving action error analysis, failure modes and effects analysis, hazards and operability analysis, and interface analysis.

SH-1 A request to retract the gears while the aircraft is on the ground.

SH-2 A failure of a constituent entity in the LGS.

SH-3 The generation of actuation commands without a request.

SH-4 The generation of a wrong actuation command.

SH-5 A failure to generate actuation commands when requested.

From the identified system hazards, the following are the identified potential LGCS Contributions to Failure Conditions (CFCs).

CFC-1 The LGCS generates actuation commands to retract the gears while the aircraft is on the ground (from SH-1).

CFC-2 The LGCS generates actuation commands when a sensor is providing invalid data (from SH-2).

CFC-3 The LGCS generates actuation commands when an EV is not functioning (from SH-2).

CFC-4 The LGCS generates actuation commands when it is in a failure state (from SH-2).

CFC-5 The LGCS generates actuation commands without a request from the pilots (from SH-3).

CFC-6 The LGCS generates opposite actuation commands to what was requested by the pilots (from SH-4).

CFC-7 The LGCS generates actuation commands for the gear retraction or extension EVs when the doors are closed (from SH-4).

CFC-8 The LGCS generates actuation commands for the door closing EVs when the gears are in transit to the requested position (from SH-4).

CFC-9 The LGCS generates self-blocking actuation commands, *i.e.* simultaneously generates actuation commands for either the door opening and closing EVs, or the gear retraction and extension EVs (from SH-4).

CFC-10 The LGCS fails to generate actuation commands after a request from the pilots and the LGCS was not in a failure state (from SH-5).

2.2.3 Development methodology

To build the LGCS requirements specification and design we used a methodology encompassing the general flow for requirements specification and design as defined in DO-178C. It consists of three major activities, as illustrated in Figure 2.4: *Develop HLRs*, *Develop Software Architecture* and *Develop LLRs*. The activities have a sequential nature as promoted by DO-178C to allow for a meticulous engineering that builds a safe product. Nevertheless, the activities have their actions organized to form iterative and incremental cycles that will gradually build their outputs until they yield a software requirements specification and design that is suitable for directing the following phases in the development process, namely coding and verification. Furthermore, as a means for quality assurance, several practitioners from avionics companies with ample experience in avionics software development and DO-178C certification participated in validating the outputs of the activities throughout the iterations.

As mentioned in previous chapters, avionics systems are routinely specified in natural language (Potter, 2012; Blouin, 2013; Schamai *et al.*, 2015). This methodology, thus, considers system requirements allocated to software (SRATS) and high-level requirements (HLRs) to be captured using natural language. Moreover, it follows the practices of the FAA's Requirements Engineering Management Handbook (Lempia & Miller, 2009) and the work from

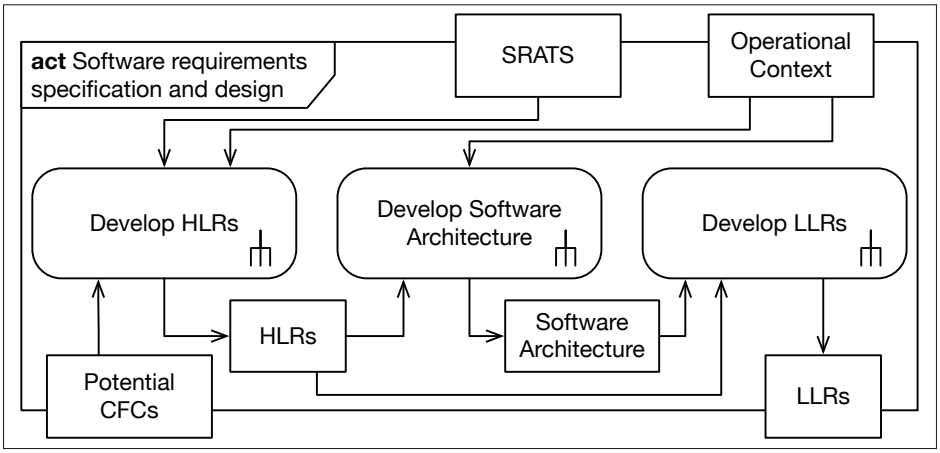


Figure 2.4 General flow for requirements specification and design. Extracted from Paz & El Boussaidi (2018).

Blouin (2013) to restrict the grammar and vocabulary used for the textual specification. The FAA handbook suggests that in order to reduce ambiguities, improve readability, and facilitate the analysis and review of the requirements, these should be specified as statements of how the software will change a set of *controllable variables* (*i.e.* values in the operational context the software can directly affect) in response to changes in a set of *monitorable variables* (*i.e.* values in the operational context the software responds to).

Once a set of system requirements is allocated to software, an operational context for the software is defined and a set of potential CFCs (identified during the software planning) has been determined, the software requirements specification, *i.e.* the *Develop HLRs* activity, begins. Figure 2.5 expands the *Develop HLRs* activity. It is necessary to first perform a manual review of the SRATS in order to verify them for ambiguities, inconsistencies or undefined conditions. If the SRATS need any clarification or correction, it is requested to the system processes and the activity waits for the clarified or corrected SRATS. SRATS may be specified with enough detail to carry on with the software design; in which case they are redefined as the HLRs. If the SRATS lack refinement or they do not preclude the CFCs, they are iteratively and incrementally developed into HLRs that accurately capture the intent of the SRATS and preclude

the CFCs. Traces between SRATS and HLRs must be established. However, if a trace cannot be established directly, then the HLR is labeled as a derived requirement.

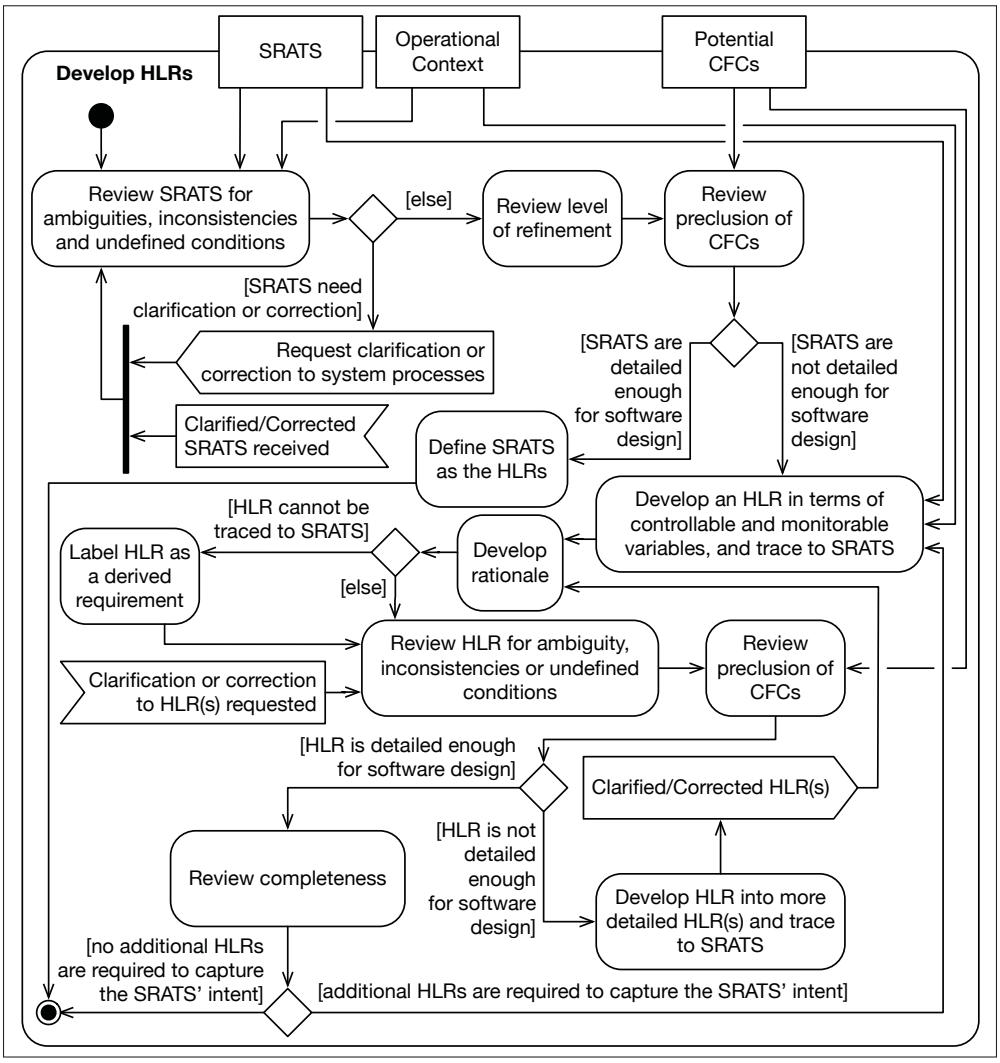


Figure 2.5 Expanded view of the *Develop HLRs* activity.
Extracted from Paz & El Boussaidi (2018).

The software design in the proposed methodology must produce design models as defined in DO-331. Hence, to maximize adherence to the guidance provided by both DO-178C, DO-331 and DO-332, the software design has been segregated as recommended by Sarkis & Dias (2014) into two different activities, namely *Develop Software Architecture* and *Develop LLRs*, that work at two distinct levels. The first level, the *Develop Software Architecture* activity rep-

resents the architecture as software components and their relationships. Software design under DO-178C guidelines must enable a black-box verification of the system's behaviour. For this to happen, DO-178C highlights the importance of avoiding introduction of complexity during the design process. Hence, the *Develop Software Architecture* activity seeks to incorporate principles of software engineering that help minimize complexity and promote verifiability, such as modularity and encapsulation (Bialy *et al.*, 2017).

Figure 2.6 expands the *Develop Software Architecture* activity. The activity starts by identifying or defining an architectural style in accordance with the expected functionality of the software to be built and its operational context. Then, the software components that will realize the functionality expressed in the HLRs, as well as the dependencies between them, are identified and defined based on such an architectural style. Each component is allocated to a subset of the HLRs. Software design patterns may be identified too as work progresses. When all responsibilities have been assigned to components and no additional components are required to cover the HLRs, the activity continues with the identification of subcomponent hierarchies realizing each of the components. Every subcomponent must be allocated to its originating HLRs, which are a subset of the ones associated to its containing component. Clarifications or corrections to HLRs may be requested to the *Develop HLRs* activity and the *Develop Software Architecture* activity waits for the clarified or corrected HLRs to be returned.

The second level, the *Develop LLRs* activity, focuses on representing low-level requirements and the functionalities that the realizing subcomponents will implement. In line with the objective of developing a benchmark avionics software specification that can be used for evaluating different model-based approaches, the *Develop LLRs* activity has two alternate definitions: one that outputs textual LLRs and another one that outputs LLRs as a DO-331 design model. Figure 2.7 expands the former and Figure 2.8 expands the latter. This also serves to illustrate how the methodology can be tweaked to adapt it to different development scenarios.

Having established the architecture initiates the work in the *Develop LLRs* activity. For every realizing subcomponent defined in the architecture, LLRs are specified in terms of the sub-

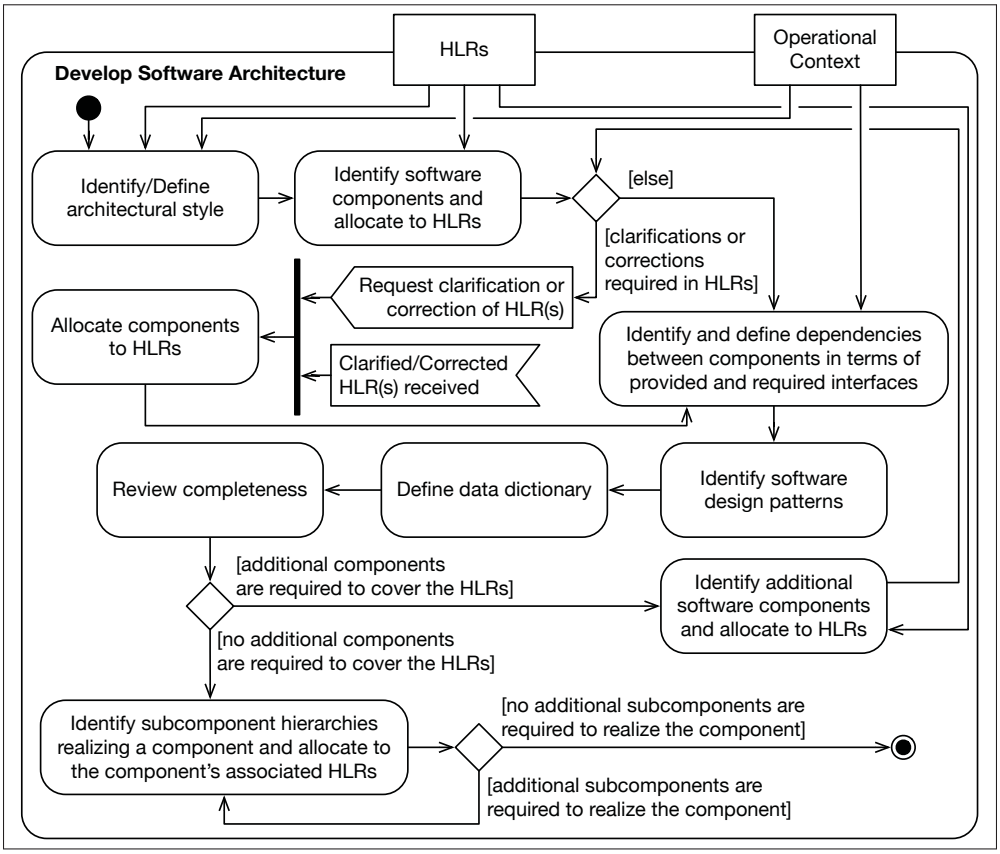


Figure 2.6 Expanded view of the *Develop Software Architecture* activity. Adapted from Paz & El Boussaidi (2018).

component's controllable and monitorable variables. Each LLR is allocated to its originating HLRs, which are a subset of the ones associated to the subcomponent. If the allocation is not possible due to required clarifications or corrections in the HLRs, these are requested to the *Develop HLRs* activity and the *Develop LLRs* activity waits for the clarified or corrected HLRs to be returned in order to continue. If the allocation is not possible because the LLR cannot be traced to HLRs, then the LLR is labeled as a derived LLR. The activity ends when source code can be directly implemented for all realizing classes without any further refinement, and all of them have their behaviour specified.

The alternate definition for the *Develop LLRs* activity that outputs a design model (see Figure 2.8) has a similar workflow. The difference is that for every realizing subcomponent de-

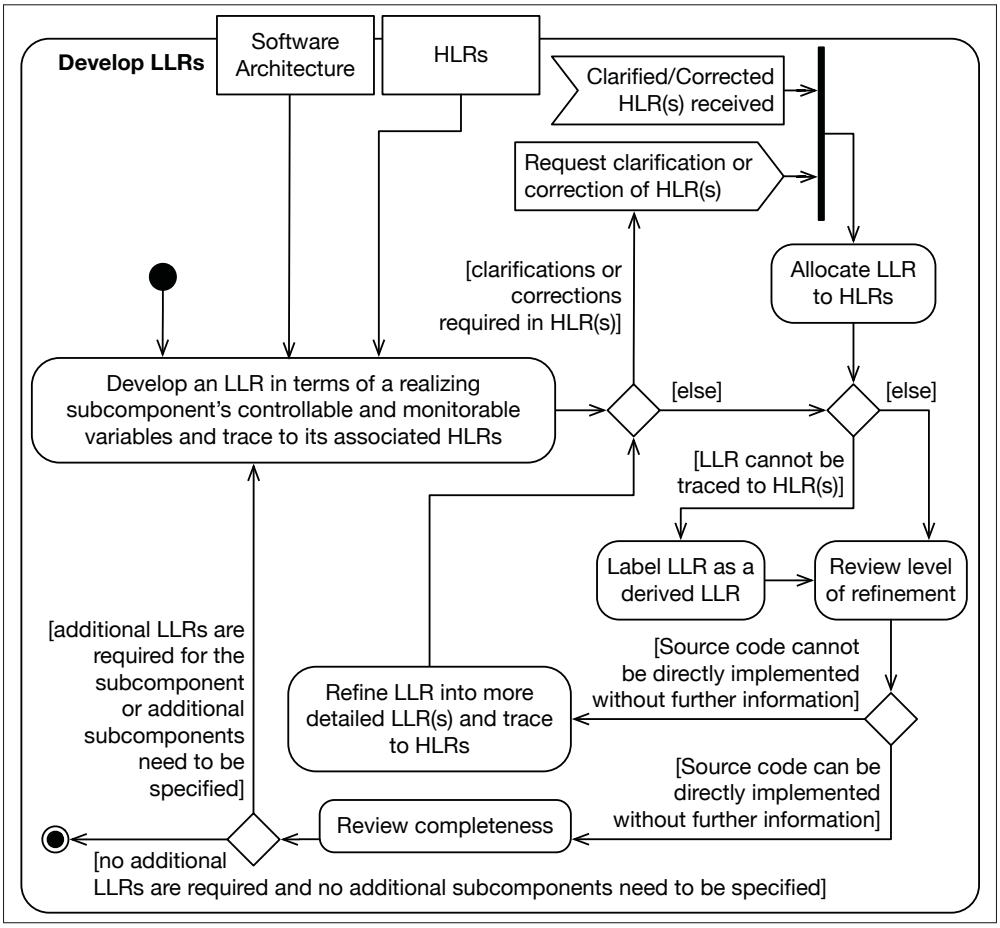


Figure 2.7 Expanded view of the *Develop LLRs* activity for specifying textual LLRs. Adapted from Paz & El Boussaidi (2018).

fined in the architecture a UML state machine or a Stateflow chart design model is created describing its behaviour. The design model's elements are allocated to their originating HLRs as well. UML state machines and Stateflow charts were chosen since this was a condition set by the industry partners. Moreover, the former are an important part of UML for modelling the behaviour of objects and the latter is commonly used by avionics providers for modelling real-time/mission critical systems Rozanski & Woods (2011); Lee (2010). DO-178C and the proposed methodology also prompted a set of essential features for the modelling of LLRs, which both of these modelling languages are able to cover: 1) layered modelling and hidden decompositions, 2) factorization of commonalities or reuse of modelled elements, 3) partial ordering and concurrent flow of control, 4) algorithms, 5) time observation and timing con-

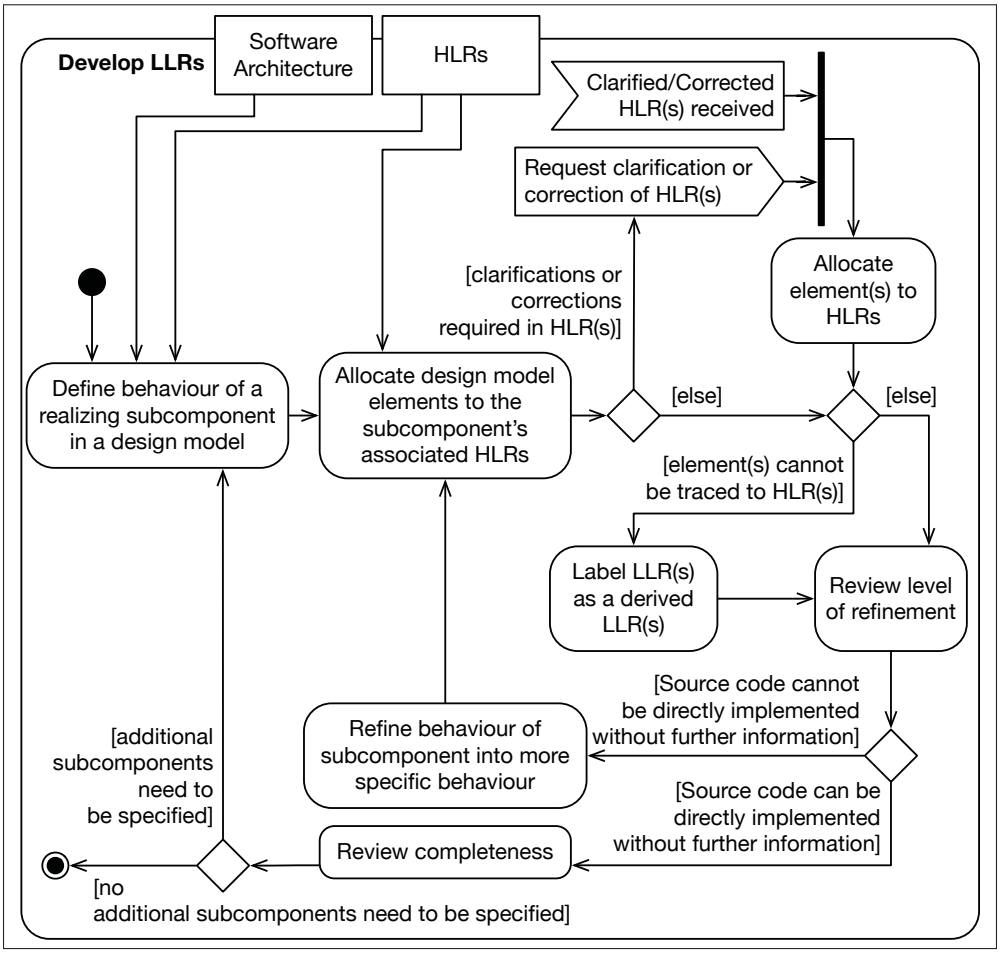


Figure 2.8 Expanded view of the *Develop LLRs* activity for specifying LLRs as design models. Adapted from Paz & El Boussaidi (2018).

straints, 6) interruptions in the flow of control and exception handling, 7) explicit interactions between distinct system parts, 8) complex trigger conditions and triggered actions, and 9) flow of data (usage, production and storage).

2.3 Software Requirements Data

Being the LGCS based on the case study presented by Boniol & Wiels (2014), the descriptions found in such a work relating to the digital controller have been taken as the SRATS. However, since the SRATS contained ambiguities, inconsistencies and undefined conditions they had to be corrected and improved before developing the HLRs. It took three iterations of SRATS and

HLRs development to obtain a favorable perception about them from the practitioners involved. Table 2.1 shows a subset of the SRATS while Table 2.2 shows a subset of the HLRs developed from these SRATS. This subset of HLRs will be used to show some of the LLRs that were developed from them (see Section 2.4).

Recall from Section 2.2.3 that requirements are specified in terms of controllable and monitorable variables. These variables are all described following the practices of the FAA's Requirements Engineering Management Handbook (Lempia & Miller, 2009). For instance, HLR-6 in Table 2.2 describes how the software will change the controllable variable `General EV Actuation Command` in response to the value of monitorable variable `Hydraulic Circuit Pressure` at a given time.

Table 2.1 Examples of SRATS for the LGCS.

ID	**Description**	**Traces**	**Precluded CFCs**
SRATS-1	When the LGCS receives data from one of the LGS sensors, the LGCS shall process three sensor readings describing the situation by a 2 out of 3 voting scheme.	HLR-1	CFC-2
SRATS-2	When the pilots switch the gear lever to `Up`, the LGS shall retract the gears in under 28 seconds.	HLR-2	CFC-1
SRATS-4	When the LGCS is currently executing either a retraction or extension sequence and a new desired gear position is received, the LGCS shall halt the current sequence and revert all the actions that were executed.	HLR-4	
SRATS-6	The LGCS shall consider prior to setting to `Open` a specific EV that 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`.	HLR-6	
SRATS-7	The LGCS shall consider prior to setting to `Open` the `Door Opening EV Actuation Command` that at least 0.2 s have elapsed since the `General EV Actuation Command` was set to `Open`.	HLR-7	
SRATS-12	When 2 s have elapsed since the `General EV Actuation Command` was set to `Open` and 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.	HLR-12	CFC-3

Table 2.2 Examples of HLRs for the LGCS. Adapted from Paz & El Boussaidi (2018).

ID	Description	Traces	Precluded CFCs
HLR-1	When the LGCS receives data from one of the LGS sensors, the LGCS shall process three Readings associated to the sensor data and describing the situation with a 2 out of 3 voting scheme.	SRATS-1 LLR-1 LLR-2 LLR-3 LLR-4 LLR-5 ...	CFC-2
HLR-2	When the LGCS receives an Up value for the Desired Gear Position, the LGCS shall carry out the retraction sequence in under 28 seconds.	SRATS-2 LLR-11 LLR-12 LLR-13 LLR-14 LLR-15 ...	CFC-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	When 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.	SRATS-6 LLR-14 LLR-43 LLR-44 LLR-45	
HLR-7	When at least 0.2 s 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	When 2 s 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	CFC-3

2.4 Design Description

2.4.1 Software architecture

The LGCS receives input from the different LGS sensors and acts upon those inputs to drive the actuation of the LGS' EVs to move the gears into the desired position. This context of operation

matches the *Process Control* architectural style (Levine, 1996), where a system uses a set of inputs to determine a set of outputs that will produce a new state of the environment. Thus, the software architecture for the LGCS follows this architectural style. Figure 2.9 presents a high-level design model of the software architecture as a UML component diagram. The figure also illustrates some of the allocations of HLRs to components. In the figure, the LGS is described as a set of modular components providing its specified functionality through a set of provided interfaces. `Sensor` components constantly take readings of their associated system entity. The `LGCS` component interfaces with these `Sensor` components and the `PilotInterface` component to pull the sensor readings and output system feedback, respectively. The `LGCS` component also continuously monitors the `PilotInterface` component to identify input of a new desired gear position. When this occurs, the `LGCS` computes and transmits the commands to the corresponding `EV` component.

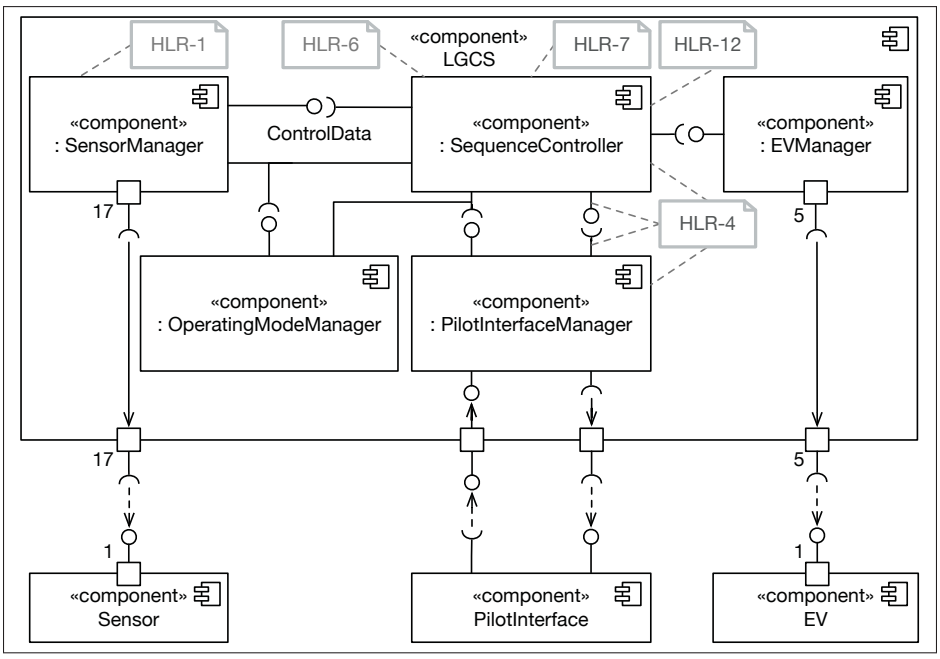


Figure 2.9 Architecture for the LGCS as a UML component diagram. Adapted from Paz & El Boussaidi (2018).

In the UML component diagram from Figure 2.9, the `LGCS` component was further decomposed into smaller components providing simpler functionality. Interfacing with the various

sensors is the SensorManager component. The SensorManager component receives the inputs from the sensors using pull communication through the interface with the corresponding Sensor component. The SensorManager component pulls data from the sensors only when requested by the SequenceController component. The SequenceController component interfaces with the SensorManager component through the ControlData interface (all other interface names are omitted to keep the figure uncluttered). The ControlData interface is detailed in Figure 2.10. The SensorManager component performs a validation of the data received from each sensor before passing it along on to the SequenceController component. The SensorManager component will report a failure of a sensor to the OperatingModeManager component when an invalid sensor data is received. The SequenceController component sequences the actuation commands according with the inputs received from the SensorManager and the PilotInterfaceManager components. The PilotInterfaceManager component will continuously monitor the PilotInterface to identify the input of a new desired gear position and, also, outputs the feedback. The EVManager component interfaces with the EVs to output the actuation commands. The OperatingModeManager component will place the LGCS into a failed state if any failure is detected, and the system will remain in such a state thereafter. All data exchange in the LGCS components' interfaces is defined in a data dictionary.

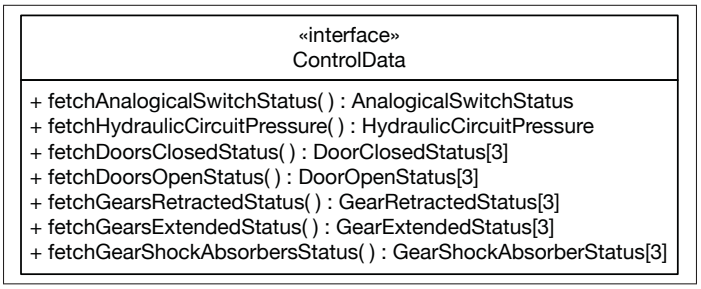


Figure 2.10 ControlData interface.
Extracted from Paz *et al.* (2020).

The LGCS was designed twice in its entirety; one time using UML and another one using Simulink. The industry partners' particular interest to have a complete design redundancy with these

modelling languages motivated this. The design with Simulink is also analogous as observed in Figure 2.11. Here, the LGCS was described as a set of subsystem blocks performing their functions over a set of inputs to produce the set of expected outputs.

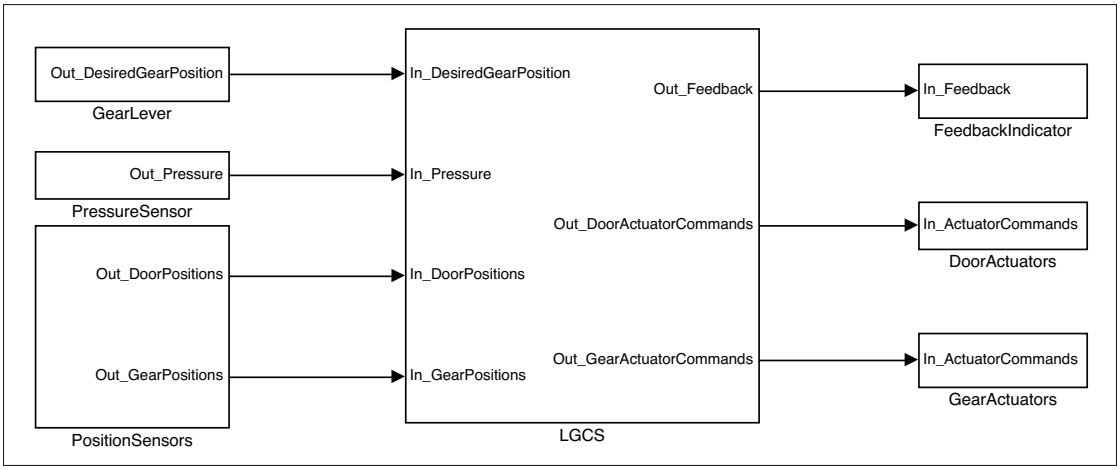


Figure 2.11 Excerpt of the architecture for the LGCS as a Simulink block diagram.

2.4.2 Low-level requirements (LLRs)

Three parallel specifications of LLRs were created, namely a textual specification using natural language and two model-based specifications using UML and Simulink (alongside with Stateflow). Table 2.3 shows a subset of the textual LLRs developed from the presented HLRs.

In the case of the design with UML, class diagrams and state machines were used to provide the detailed design of the subcomponents. An example is given in Figure 2.12. It shows a fragment of an equivalent model-based specification as a UML state machine for LLR-35, LLR-43, LLR-44, LLR-45, LLR-56 and LLR-57. The state machine fragment represents the state to which the `SequenceController` component transitions into after transmitting a command to the general hydraulic valve to pressurize the hydraulic circuit. The component will remain in such a state until the `hcp` (hydraulic circuit pressure) value is within the system's operating range. Once the `hcp` is within the specified range, the component transitions into another state. If the `hcp` does not reach an accepted value within an allotted time (2 s) a transition into a failure state

Table 2.3 Examples of LLRs for the LGCS. Extracted from Paz & El Boussaidi (2018).

ID	Description	Traces	Component
LLR-14	If the PressurizationEvent and the DelayDOEVActuationTimeoutEvent are raised, the Door Opening EV Actuation Command shall be set to Open and the waitForDoorsOpen method shall be activated.	HLR-2 HLR-3 HLR-6 HLR-7	Sequence-Controller
LLR-35	If the waitForHydraulicPressure method is active and the RevertEvent is raised, all the actions that were previously executed shall be reverted.	HLR-4	Sequence-Controller
LLR-43	If the General EV Actuation Command is set to Open, the waitForHydraulicPressure method shall be activated.	HLR-6	Sequence-Controller
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	Sequence-Controller
LLR-45	If the waitForHydraulicPressure method is active and the overall value of the Hydraulic Circuit Pressure is greater than or equal to 30,000 kpa and less than 35,000 kpa, the waitForHydraulicPressure method shall end and the PressurizationEvent shall be raised.	HLR-6	Sequence-Controller
LLR-46	If the General EV Actuation Command was set to Open, the DelayDOEVActuation method shall be activated.	HLR-7	Sequence-Controller
LLR-56	If the waitForHydraulicPressure method is active and 2 s have elapsed since the General EV Actuation Command was set to Open, the PressurizationTimeoutEvent shall be raised.	HLR-12	Sequence-Controller
LLR-57	If the waitForHydraulicPressure method is active, the PressurizationTimeoutEvent is raised and the overall value of the Hydraulic Circuit Pressure monitorable variables is less than 30,000 kpa, the FailureEvent shall be raised.	HLR-12	Sequence-Controller

is taken instead. In the event of a reversion, a transition into a reversion state is taken. The allocation of HLRs to elements of the UML state machine fragments in the figure is done by means of UML comments.

The LGCS subsystem block, shown in Figure 2.11, was decomposed into smaller, interconnected subsystem blocks. Figure 2.13 presents this decomposition. The detailed design is provided through Stateflow charts. Figure 2.14 presents an excerpt of the Stateflow chart for the SequenceController Simulink block (equivalent of Figure 2.12).

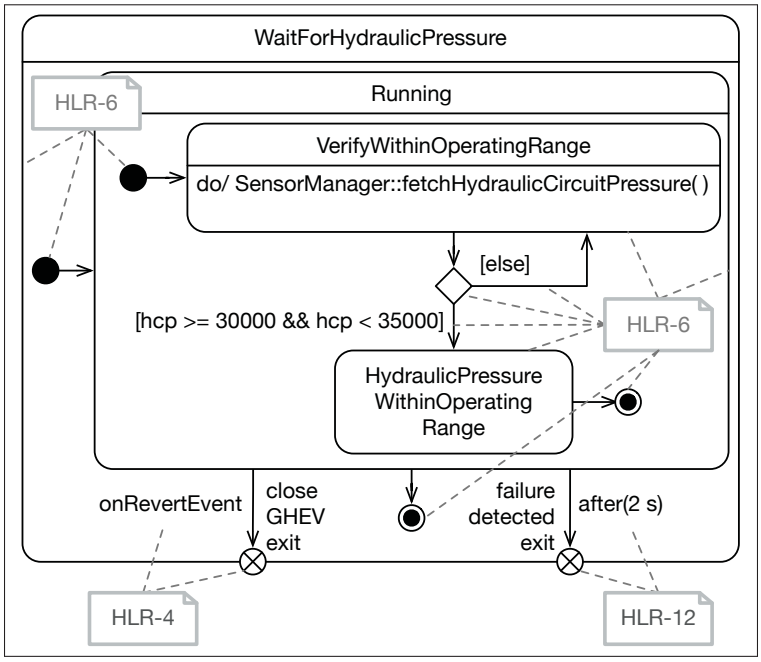


Figure 2.12 Excerpt from the UML state machine associated to the SequenceController component. Adapted from Paz *et al.* (2020).

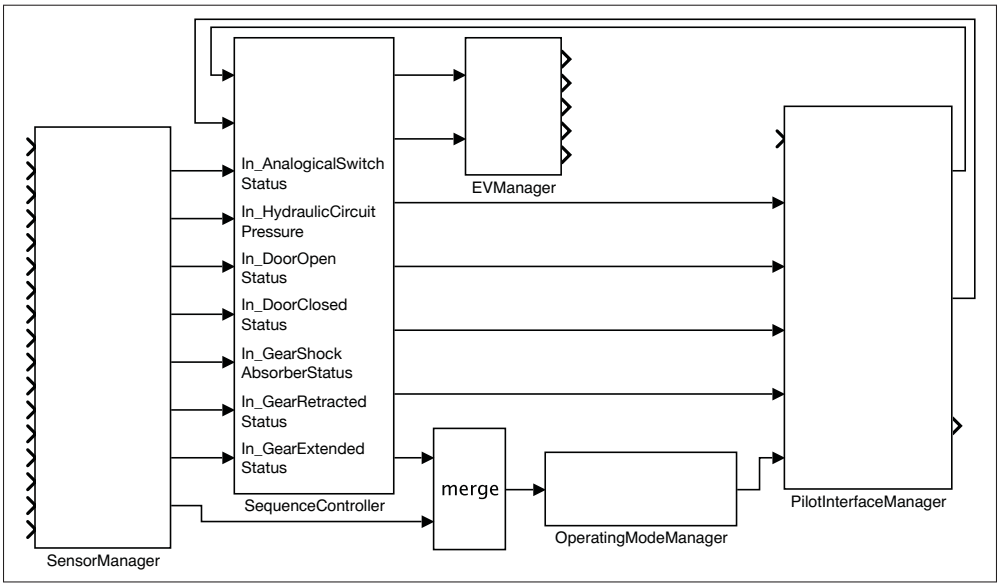


Figure 2.13 LGCS decomposition as a Simulink block diagram. Extracted from Paz *et al.* (2020).

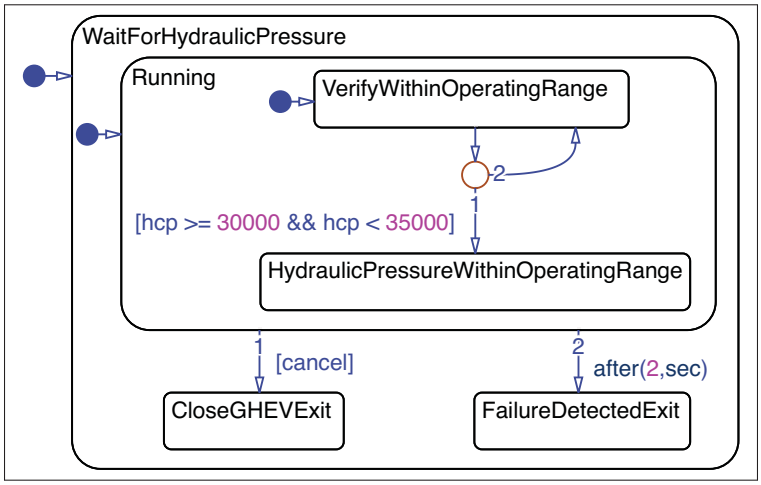


Figure 2.14 Excerpt from the Stateflow chart realizing the `SequenceController` subsystem block. Extracted from Paz *et al.* (2020).

2.5 Discussion

This section discusses and elaborates on some lessons learned, and challenges and issues experienced for each of the activities performed to build this requirements specification and design. This section also highlights some limitations of the documentation and outlines steps to overcome them and extend it.

2.5.1 Requirements specification

2.5.1.1 Quality and granularity of SRATS

As mentioned in Section 2.3, the SRATS were taken from the descriptions by Boniol & Wiels (2014). These descriptions, however, were scattered throughout the text, which hindered the task of developing the HLRs and establishing traceability links between the SRATS and HLRs. Furthermore, the SRATS' review in the *Develop HLRs* activity, found them to contain several inconsistencies, ambiguities and confusing wording. Hence, the SRATS had to be corrected, clarified and uniquely identified first before proceeding with the development of the HLRs.

The heavy reliance on review actions in the *Develop HLRs* activity is intended to guarantee the requirements are clear. It is imperative that both sets of SRATS and HLRs are reviewed to be at an acceptable quality level.

While working on the SRATS, it was observed that these can be, in fact, very detailed so as to guide the software's design without any further refinement into HLRs. Feedback on the matter from the practitioners involved points out that such a situation is not infrequent in the industry because the SRATS are intended to give the software development team everything they need to know to develop the software. The more clear, precise and complete the SRATS are, the easier the job becomes for the development team. In the LGCS, the SRATS were not developed to a refinement level that could lead directly to software design, as it was sought to have a separation between the SRATS and the HLRs in order to make the documentation as rich as possible.

2.5.1.2 Requirements specification language

Practitioners from industry routinely specify SRATS and HLRs using natural language in Microsoft Office Word documents or Microsoft Office Excel spreadsheets and manage them with the help of textual requirements databases like DOORS (Potter, 2012; Blouin, 2013). Thus, in order for this LGCS documentation to serve as a benchmark specification, the SRATS, HLRs and LLRs of the LGCS were specified in natural language. However, due to the inherent ambiguities of natural language (Moy *et al.*, 2013), a form of controlled natural language was used for their specification. As suggested by Lempia & Miller (2009), SRATS, HLRs and LLRs were written using the names of monitorable and controllable variables, and system and software architectural elements. Having the requirements formulated in this way brought important advantages in terms of the properties of a good requirements specification (*e.g.*, consistency, completeness). Subsequent stages of the development of the LGCS may also see benefits from this form of specification such as the generation of test cases. Our approach addresses the creation of specification models using a proposed requirements modelling language to handle these concerns for HLRs from a model-based perspective in the context of DO-178C.

2.5.2 Design

2.5.2.1 Design modelling language

Representing the LLRs in UML state machine models posed its challenges as they stressed our knowledge and understanding of the UML notation and its semantics, and the notation and semantics themselves. The UML specification has been criticized for its omissions, inconsistencies, vaguenesses and comprehensibility problems in the UML metamodel and in the definition of its semantics (Fecher *et al.*, 2005). Hence, a major lesson here is the UML specification should not be taken on its own as a design standard for design models under DO-178C, DO-331 and DO-332. It is necessary that the design standard defines any additional information, constraints and specificities that help avoid any notational and semantical unclarities of UML. For instance, one design guideline that emerged in discussion workshops with industry partners was related to the amount of layered substates in a UML state machine and Stateflow chart. These languages do not define a limit to the number of layered substates that may be added in a design model, which creates comprehensibility and verifiability issues. Furthermore, a long overdue improvement of UML state machines surfaced during the *Develop LLRs* activity. UML state machines capture trigger conditions and triggered actions as annotations on the transition arrows. The notation lacked scalability when developing some of the LLRs (*e.g.*, LLRs developed from HLR-1). Herrmannsdörfer & Berenbach (2008) have suggested the combination of diagrammatic and tabular notations to facilitate the specification and analysis of such complex trigger conditions. Nevertheless, this requires extending UML with non-standardized constructs.

2.5.2.2 Granularity of LLRs

In the *Develop LLRs* activity, it was observed that the operating context of the LGCS originated a fair amount of intertwined conditions that the software had to respect at any given moment. Therefore, developing the LLRs for the LGCS with an appropriate level of granularity was challenging. DO-178C expects LLRs to be very detailed in order to enable source code to

be implemented without the need for more information. This may be misleading towards wanting to express conditions as close as possible to the code, say in pseudocode or in an action language (*e.g.*, UML Alf). Unlike the situation with SRATS and HLRs where these could be considered the same, DO-178C requires a greater separation between LLRs and code than pseudocode / action languages are able to provide. The reason lies with the need to enable black-box verification of the software's behaviour.

2.5.2.3 Bidirectional traces in model-based LLRs

Both the *Develop Software Architecture* and the *Develop LLRs* activities are required to allocate HLRs to the elements of their resulting design models. In the case of UML this was achieved through the use of UML comments. Sarkis & Dias (2014) recommend maintaining such traceability links in the design models themselves and that the use of comment blocks is sufficient for the task. This certainly works for supporting backwards traceability, *i.e.* linking LLRs to their source HLRs, when the latter are uniquely identifiable. However, the same cannot be said for forwards traceability, *i.e.* linking HLRs to their developed model-based LLRs, which is also required by DO-178C. In order to enable requirements-based verification of the software, and give proper visibility to requirements when verifying coverage and assessing the impact of change, all requirements need to be uniquely identifiable. Identifying what can be an LLR in UML state machines is not trivial and not possible without an extension to UML. Two possible solutions were identified through discussion workshops with the industry partners. One solution is to consider an LLR as a 4-tuple made up of an origin state, a destination state, a transition from the origin state to the destination state, and a set of actions triggered and completed during such a transition. Such a solution limits the use of the notation. However, the definition would become more complex if fork, join and choice pseudostates are taken into account. Another solution, in an attempt to avoid a complex definition like the one just mentioned, is to consider an LLR to be a complete UML state machine. This comes with its own set of issues, especially related to the separation between LLRs and HLRs. The same issue may appear also in specification models capturing non-text-based HLRs.

2.5.2.4 Consistency of heterogeneous design models

Effectively designing the LGS requires a complex modelling approach that can cope with 1) dealing with diverse components, including mechanical, electronic and software, each one of these with its own underlying theories and domain vocabularies, and 2) dealing with various aspects of the same component, such as their function, structure and behaviour. It is already hard enough to relate information presented in different model views, *e.g.*, in software, linking states and transitions of a state machine to classes and methods, or system functions to packages, classes and methods. It is even harder to relate information that is spread across multiple models and expressed in different modelling languages (Lee, 2010; Eker *et al.*, 2003; Yu *et al.*, 2011; Combemale *et al.*, 2014; van den Brand & Groote, 2015). In the LGS, for instance, the hydraulic electro-valves (EVs) exist as 1) a physical object, 2) a mechanical engineering abstraction governed by mathematical equations, and 3) a software abstraction manipulated by the control software running in the digital controller. Ensuring consistency between heterogeneous design models is an important problem, in and of itself. Adding to that is that safety-critical avionics systems are highly regulated and, thus, their development must adhere to stringent quality and verification norms, like DO-178C presented in Chapter 1.1.

It is possible that inconsistencies are introduced during the design process of a given safety-critical software since the different design models created are likely developed independently. This can be mitigated with the use of design standards. In the case of the LGCS, some inconsistencies were deliberately inserted in the design models presented in the previous sections. In virtue of DO-178C certification, evidence must be gathered to demonstrate the design models are consistent and conform to design standards. The first step in ensuring consistency is determining that each model, taken separately, was built in accordance with the established design standards. These usually comprise the methods, notations, rules, constraints, guidelines, and conventions defined in the modelling language specifications and are most likely enforced by default in the modelling environment. However, they can also include particular rules, guidelines and conventions that further instruct on, or constrain, the use of certain language constructs according with company-specific practices. Moreover, they can even provide ways

of using available constructs to represent certain abstractions for which a modelling language does not define a precise construct.

The next step is to analyze the different models in search for correspondences between their elements. Notice that this step requires a pre-established knowledge on how the models themselves are related. For example, in the LGCS design models presented, there is a close similarity between the transition in Figure 2.12 from the `Running` state to the `close GHEV exit` point and the transition in Figure 2.14 from the `Running` state to the `CloseGHEVExit` state. The final step is to analyze the identified correspondences to decide if they exhibit the same features and behaviour and conform to design standards. For example, the two transitions in the previous correspondence are inconsistent. The trigger on the UML transition is expressed as an event name (*i.e.* `onRevertEvent`), while the equivalent transition trigger in the Stateflow chart is expressed as a condition on an input Boolean variable called `cancel`. The analysis indicates that these transitions must be reviewed and the inconsistency fixed accordingly. All these steps must be performed manually, in the absence of tools, and for every construct used in the design models.

Establishing model consistency and adherence to design standards are resource-consuming and error-prone activities. While automated design verification and validation tools can help, they cannot be used in isolation. Indeed, if the output of such tools is not manually verified by a human being, then the tools themselves need to be *qualified*, *i.e.* they need to be subjected to the same—if not a higher—level of scrutiny as the systems they are meant to verify Varró (2016). It is, therefore, essential to devise a *hybrid* approach where automated tools aid engineering teams in flagging errors for review and eventual correction, only to be followed by a more traditional simulation and testing process.

2.6 Chapter Summary

This chapter presented a detailed requirements specification and design for a landing gear control software (LGCS). This documentation was developed and organized according with DO-

178C guidelines. The chapter also discusses some issues encountered during the development of the LGCS specification and design. The elements of the LGCS presented in this chapter will be used as a running example in the rest of this thesis to illustrate the other contributions. It is to be noted that the descriptions given in this chapter are simplified to provide better focus for the context of this thesis. The complete version of the specification and design is available in Appendix II of this dissertation. The documentation has also been made available online (Paz & El Boussaidi, 2017) for it to be used by other researchers and practitioners as a benchmark for supporting the evaluation of different (existing and new) model-based approaches.

CHAPTER 3

CHECSDM: CONSISTENCY OF HETEROGENEOUS EMBEDDED CONTROL SYSTEM DESIGN MODELS

This chapter presents *checsdm*[1] (Consistency of Heterogeneous Embedded Control System Design Models), a systematic approach, based on MDE, for assisting engineering teams in ensuring consistency of heterogeneous design of safety-critical avionics systems and supporting evidence-gathering efforts for certification. In this regard, *checsdm* ensures that 1) for each modelled element appearing in more than one design model, the element exhibits the same properties and behaviour, and 2) the design models conform to *design standards*. Recall design standards specify methods, notations, rules, constraints, guidelines and conventions to be used in the development of the design models. They can be specific to the company or inferred from a given regulation such as DO-178C. Three features of *checsdm* can be highlighted. First, it aims to cover more design scenarios than existing related approaches by providing a design-scenario-independent framework to support verification of heterogeneous design models. Second, it provides facilities that ease the verification of model consistency and adherence to design standards. Third, it enforces specific aspects of DO-178C compliance needs (*e.g.*, explicitly defining a design scenario and design guidelines, ensuring design consistency) in an effort to aid the recollection of evidence for certification.

The chapter is organized as follows. Section 3.1 describes in detail the *checsdm* approach. Section 3.2 describes a concrete instantiation of *checsdm* using the LGCS and a mix of UML, Simulink and Stateflow design models.

3.1 The *checsdm* Approach

checsdm is developed as a generic *methodology* and a *tool framework*, which can be applied to various scenarios involving different modelling languages and different design guidelines. The methodology comprises an iterative three-phased process (see Figure 3.1). The first phase

[1] Pronounced "checks them".

is the *elicitation phase*, which consists in specifying the *requirements* of the design scenario at hand in terms of: 1) determining the mix of modelling languages that are going to be used and how to use them depending on the system's nature and the languages' purposes, 2) identifying the mapping rules between the different modelling languages, 3) defining intra-model design guidelines, *i.e.* design guidelines specific to models in each language taken separately, and 4) defining inter-model design guidelines, *i.e.* design guidelines that concern cross-model constructs. The second phase is the *codification phase*, which consists in using meta-level functionalities of the proposed tool framework to codify, as required: 1) metamodels of the various modelling languages used, 2) the mapping rules between the modelling languages, 3) the intra–model design guidelines, and 4) the inter-model design guidelines. *checsdm*'s third phase is the *operation phase*, where a toolchain derived from the previous phase is applied to actual system designs. This phase can help refine the mapping rules and guidelines or identify new ones. Instantiations of the *checsdm* approach, using their defined mapping rules and design guidelines, help support evidence-gathering efforts that will show achievement of objectives and activities for regulatory certifications. The following subsections describe in detail the three phases, including the tools and technologies used to support the *codification* and *operation* phases.

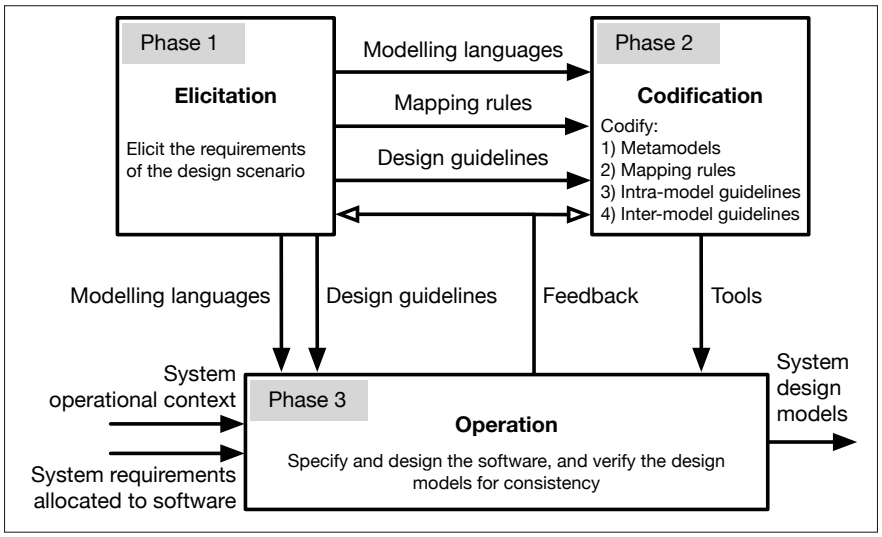


Figure 3.1 General flow of the *checsdm* approach. Adapted from Paz *et al.* (2020).

3.1.1 Elicitation phase

Four steps make up the *elicitation* phase:

1. **Determination of the mix of modelling languages.** The combination of modelling languages that are going to be used is determined based on the type of system at hand and the languages' purposes.

2. **Identification of mapping rules.** Correspondences between the constructs of the different modelling languages are identified and mapping rules are established.

3. **Definition of intra-model design guidelines.** Design guidelines specific to models in each language are introduced to direct the creation of the individual models.

4. **Definition of inter-model design guidelines.** Design guidelines that concern cross-model constructs are introduced to instruct on elaborated syntactical and semantical relations between the modelling languages.

Eliciting requires a careful analysis of the modelling languages' specifications to acquire a proper understanding of the type of information they can represent and how to do so. Tools to help with the elicitation steps are described in the following.

3.1.1.1 Mix of modelling languages

Heterogeneous design scenarios use different mixes of modelling languages. Such scenarios can be characterized according with 1) the degree of coverage of system elements by the design models, 2) the perspectives covered by the design models, 3) the level of abstraction at which the models represent the design, and 4) the degree of overlapping between the elements of the design models. Figure 3.2 presents this characterization in a feature diagram defining four dimensions: 1) coverage of system elements (partial, complete), 2) design perspective (structural, behavioural), 3) level of abstraction (same, different), and 4) overlap of system elements

(partial, complete). The dimensions are cloneable features with a minimum cardinality of 2, since at least two modelling languages are expected to be used in heterogeneous design.

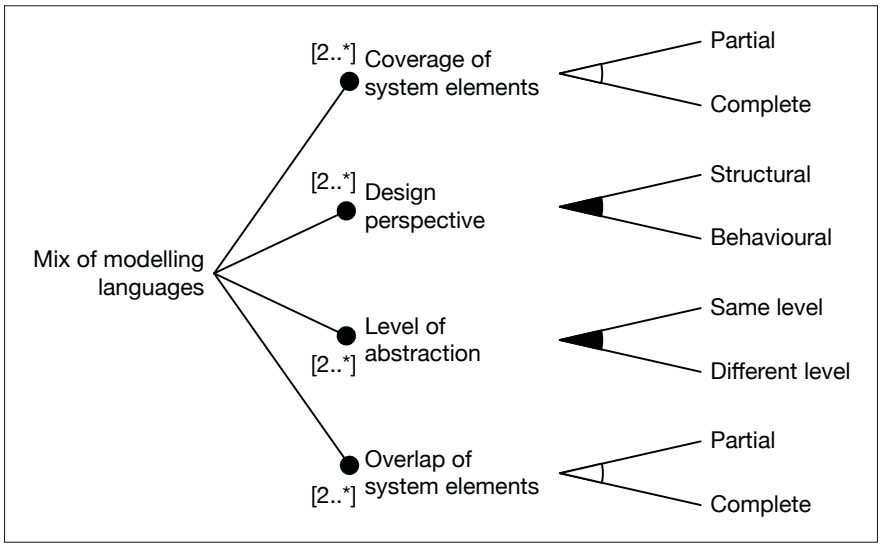


Figure 3.2 Feature model characterizing the mix of modelling languages. Extracted from Paz *et al.* (2020).

A design scenario, *i.e.* a configuration of the mix of modelling languages, needs to be created and input to the following phases. Over twenty-two scenarios can be derived from the possible feature combinations. Some examples of common scenarios taking UML and Simulink as the modelling languages are as follow:

- Simulink to represent a high-level, structural view of the elements that make up a system, and UML to represent the structural and behavioural aspects at a lower-level of abstraction for a particular element of the system that will be implemented on software. This combination will result in a partial overlap of elements at different levels of abstraction.

- In a system that handles both continuous-time and discrete-time inputs, Simulink can be used to represent the continuous-time parts of the system while UML can focus on describing the discrete-time parts. This combination will result in a partial overlap of elements, where two models represent parts of the system, and capture structural and behavioural aspects at the same levels of abstraction.

It is to be noted that developing design models with complete overlap and coverage of system elements is not a desirable scenario due to the high cost this task implies. Still, a complete design redundancy can be beneficial, for instance, in multiple version dissimilar software and in other cases, for instance, when the complete description of an element is scattered throughout several design models (*i.e.* every design model represents a partial view of the element) (Lee, 2010; Ferrari *et al.*, 2013; van den Brand & Groote, 2015).

3.1.1.2 Mapping rules

Mapping rules represent requirements on the relationships between the constructs of the different modelling languages. More specifically, a mapping rule relates model constructs from two or more modelling languages of interest (*e.g.*, UML and Simulink). The mapping rules must cover the different possible design perspectives and levels of abstraction provided by the modelling languages. The mapping rules are documented using the metamodel in Figure 3.3. Each mapping rule is given an identifier (ID) and a name for the relationship between the two model constructs. The set of *when* clauses specifies the conditions under which the current relationship holds. The set of *where* clauses specifies the relationships (*i.e.* other mapping rules) that must also hold whenever elements are participating in the current relationship. As a result, *where* clauses may cause the propagation of inconsistencies. Mapping rule codification is explained as part of the *codification* phase (see Section 3.1.2). Examples of mapping rules are given with the motivating example in Section 3.2.1.

When the mapping rules are executed, a mapping model must store the identified correspondences in order to make them explicit. A mapping metamodel is defined for such a purpose. In addition, the mapping metamodel includes facilities to flag consistency issues between the design models. Figure 3.4 presents this metamodel. The `MappingModel` comprises a collection of `Mappings`. A `Mapping` relates two or more model elements (`elements`) belonging to two or more given design models. If both elements are consistent, according with the applicable `mappingRule`, then the `matching` flag is set to *true*, otherwise it is set to *false*. In this way consistency issues can be flagged.

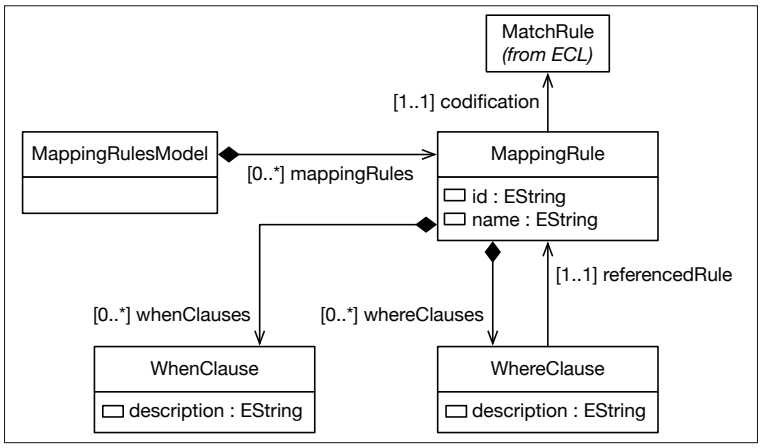


Figure 3.3 Mapping rules metamodel.
Extracted from Paz *et al.* (2020).

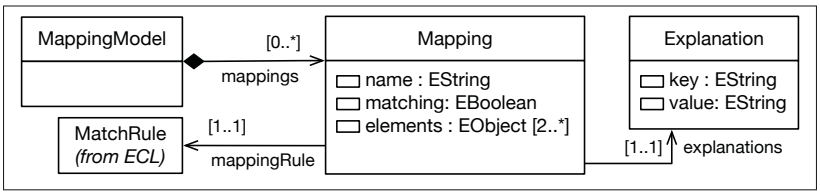


Figure 3.4 Mapping metamodel.
Extracted from Paz *et al.* (2020).

Mappings between two model elements, whether they are matching or not, are given some explanations containing the reasons for the obtained result. The `explanations` attribute is responsible for capturing explanations. Explanations contain the mapping rule's failed *when* or *where* clauses. Explanations come in handy when it is time to review two elements that were mapped but were flagged to be inconsistent. An example of a mapping model is presented later with the motivating example in Section 3.2.

3.1.1.3 Design guidelines

Design guidelines are intended to direct engineers on the use of the diverse modelling languages. Since engineers usually work in an independent fashion, the guidelines help ensure consistency between the design models they create. Guidelines are documented using the

metamodel in Figure 3.5, which is derived from the textual template used for the MAAB style guidelines (MathWorks Automotive Advisory Board, 2012). Each guideline is given an identifier (ID), a title and one of two possible priorities: *mandatory* or *recommended*. The description and the rationale fields provide the textual narrative for the guideline and its justification, respectively. A guideline is scoped to one or more modelling languages to which its guidance applies. This will categorize the guideline as intra-model or inter-model, respectively. A guideline may reference other guidelines as prerequisites or as referenced guidelines providing additional details where applicable. Guideline codification is explained as part of the *codification* phase. For a design model to be regarded as compliant with the design guidelines, all mandatory guidelines must be met while recommended guidelines can either be met or be subject to a deviation. In the latter case, the deviation must be properly justified and documented. Examples of design guidelines are given with the motivating example in Section 3.2.1.

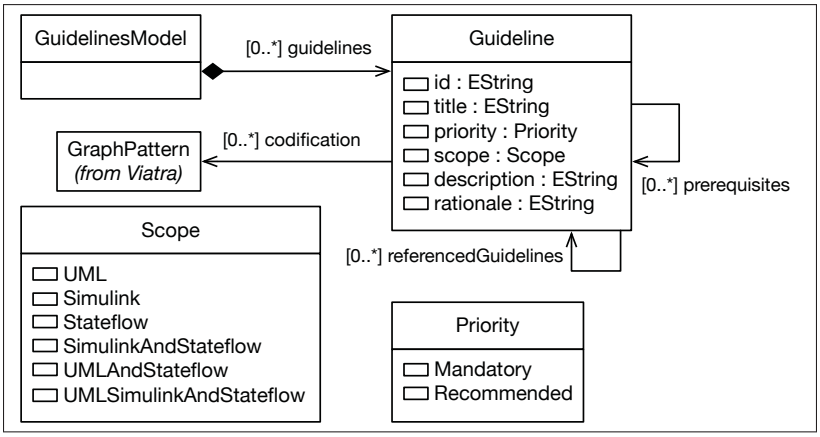


Figure 3.5 Guidelines metamodel.
Extracted from Paz *et al.* (2020).

3.1.2 Codification phase

The *codification* phase consists in using the proposed tool framework to derive a toolchain that will assist engineering teams in ensuring consistency of the heterogeneous design models during the *operation* phase. Figure 3.6 presents the high-level components of the tool framework.

The framework is based on the Eclipse platform and its modelling technologies. Three steps make up the *codification* phase:

1. **Metamodelling.** Metamodels representing the various modelling languages of the design scenario are created using the Eclipse Modeling Framework (EMF) (The Eclipse Foundation, 2017a). This step also involves developing custom model importers to transform original model files into their EMF representations.

2. **Codification of mapping rules.** Mapping rules are codified using the Epsilon Comparison Language (ECL) and Epsilon Object Language (EOL) (Kolovos *et al.*, 2018).

3. **Codification of design guidelines.** Design guidelines (intra-model and inter-model) are codified using the Viatra Query Language (VQL) (The Eclipse Foundation, 2017c).

Developer's and user's guides for *checsdm* are available in Appendix IV and online (Paz & El Boussaidi, 2019f).

3.1.2.1 Metamodelling

Modelling languages are codified as Ecore metamodels using EMF (The Eclipse Foundation, 2017a). EMF has been a successful modelling framework used for many modelling technologies in both academic and industrial contexts. EMF together with the Eclipse platform, provide an entire ecosystem and tooling for model-driven engineering. EMF provides metamodels for commonly used modelling languages, like UML and SysML. EMF-based metamodels for many other modelling languages can be downloaded and added into Eclipse via plug-ins. In the event no existing metamodel is found for a particular modelling language, EMF supplies all the tooling necessary to create the metamodel. This activity requires a careful analysis of the modelling language's specification in order to develop a technically valid metamodel (*i.e.* do not contravene the language specification). If a language specification is not openly available, a technically valid metamodel may have to be reverse-engineered from specifications. A custom

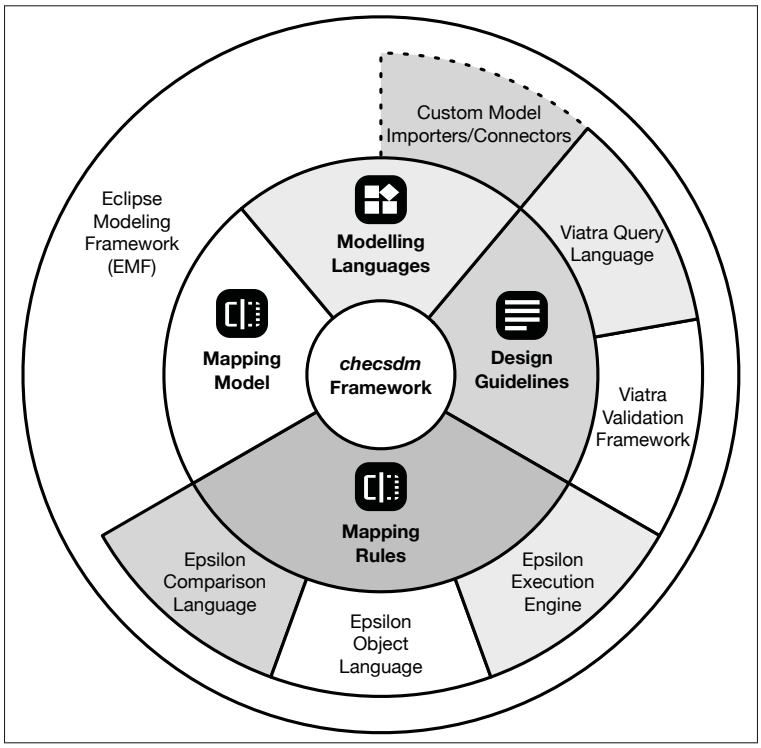


Figure 3.6 The *checsdm* tool framework.
Extracted from Paz *et al.* (2020).

model importer tool, transforming from the original format into its equivalent EMF model, will need to be codified as well.

Metamodelling requires that all aspects of the language for which a metamodel is being created are properly captured. How such a metamodel is defined depends on the expertise of the modeller. *checsdm*'s feedback loop helps perform this task iteratively. If a language's metamodel is not properly capturing all of its aspects, certainly problems during the operation phase will arise and, therefore, corrections will have to be made.

3.1.2.2 Codification of mapping rules

Mapping rules are codified as `MatchRules` using ECL (Kolovos *et al.*, 2018). Each mapping rule is, therefore, associated to a set of ECL `MatchRules`, as defined in the mapping rules metamodel (see Figure 3.3). These rules are described at the metamodel (*i.e.* modelling lan-

guage) level. ECL enables the specification of rule-based model comparison algorithms for identifying overlapping elements in homogeneous and heterogeneous models (Kolovos *et al.*, 2018). ECL comparison algorithms are expressed using EOL statements. EOL allows the definition of context-dependent operations that can be called on instances of the types in the input metamodels as if they were natively defined by the types. This technique provides a significantly high readability of the codifications and keeps them close to their natural language specifications.

When Epsilon's rule execution engine executes the `MatchRules`, it stores the identified correspondences in a mapping model conforming to the defined mapping metamodel (see Figure 3.4). Facilities for viewing and manually editing mappings between the design models' elements are provided by a mappings editor generated by EMF from the mapping metamodel. Screenshots can be seen in Section 3.2.3.

ECL produces the cross product between all element instances of the input design models that coincide with the element types defined in the mapping rules' signatures. However, a mapping rule will only return a successful mapping of those instance elements that strictly meet the conditions defined by its *when* and *where* clauses. The degree of fuzziness employed for these clauses is left to the users of the *checsdm* approach. Furthermore, not all mappings—whether they are flagged as consistent or inconsistent—in the resulting mapping model will indeed be consistent or inconsistent, and could be intentional. As a result, heuristics for cleaning the mapping model should be codified as part of this step. An example of an heuristic is given in Section 3.2.2.

3.1.2.3 Codification of design guidelines

Design guidelines must be codified in a way that expresses well-formedness constraints at the metamodel (*i.e.* modelling language) level. This is achieved by codifying them as `GraphPatterns` using VQL (The Eclipse Foundation, 2017c). VQL is a graph pattern-based language that provides a concise, declarative syntax to specify structural model queries. Each design guide-

line is associated to a set of `GraphPattern` queries, as defined in the guideline metamodel (see Figure 3.5). Furthermore, the `GraphPatterns` must be annotated with the `@Constraint` annotation. These annotations are processed by the Viatra Validation Framework, which will automatically generate facilities to 1) link the `GraphPattern` to the given Eclipse editor ID, 2) initiate the execution of `GraphPatterns` on EMF instance models opened with the linked Eclipse editors and, 3) upon violations of the constraints, handle the creation and display of markers to the user in the Eclipse Problem View. Viatra was chosen for its scalability, *i.e.* its capacity for evaluating queries on very large size models (Varró, 2016).

3.1.3 Operation phase

The *operation* phase covers the software specification and design processes, and the part of the verification process that is related to the verification of outputs from the design process. Figure 3.7 illustrates the flow of this phase. Six steps make up the *operation* phase:

1. **Software specification.** The software's HLRs are specified from the given SRATS. To support this step we introduce SpecML, presented in Chapter 4.

2. **Software design.** The software is designed from a given operational context and the set of HLRs from the previous step following the mix of modelling languages and design guidelines established during elicitation.

3. **Verification of intra-model design guideline compliance.** The resulting design models are individually verified for intra-model guideline compliance.

4. **Mapping of design models.** Correspondences between the design models are identified and mappings are established between overlapping elements.

5. **Verification of inter-model design guideline compliance.** Using the mappings of the previous step, the design models are verified together for inter-model guideline compliance.

6. **Review and resolution of violations and consistency issues.** The flagged guideline violations and consistency issues are examined and handled accordingly.

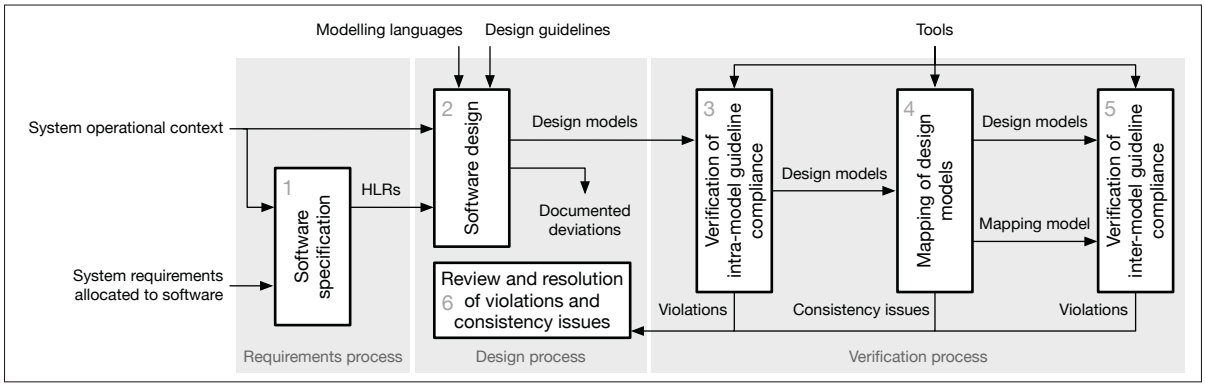


Figure 3.7 Flow of the *operation* phase. Adapted from Paz *et al.* (2020).

Step 1 is the development of the software specification, *i.e.* the HLRs. This step is described in the following chapter (Chapter 4) with our proposed requirements modelling language, SpecML. We focus in this chapter on software design and its verification. Step 2 is the creation of the software design. This comprises both the architecture and the detailed design. Design processes often vary from one organization to another, thus, it is considered a generic step that must be duly adjusted with company-specific activities. However, two inputs coming from the *elicitation* phase are necessary prior to the start of the design process: the modelling languages that are to be used and the established design guidelines (both the intra- and inter-model). Two more inputs are required. These are dependent on the actual system to be designed: the system's operational context and the HLRs. Software design models are developed from the set of HLRs taking into account the given operational context for the system. They must be developed following the established mix of modelling languages and design guidelines.

It is possible that guideline violations and consistency issues may have been introduced during the software design process since the different design models are likely developed independently following particular practices. Steps 3 through 5 automatically verify the outputs of the design process and ensure their consistency. These steps make use of the tools built during the *codification* phase. The verification of design guideline compliance is split into two

steps. Step 3 verifies compliance of intra-model design guidelines, *i.e.* design guidelines that are scoped to a single modelling language. Step 4 involves the analysis of the design models in search for correspondences between their elements and identifying overlapping elements that must be consistent. The analysis results are stored in the mapping model. Step 5 verifies compliance of inter-model design guidelines, *i.e.* design guidelines that concern cross-model constructs, after the overlapping elements have been identified.

A feedback loop exists from each one of these steps to the design process when an analysis of the design models indicates a guideline violation or a consistency issue occurred and needs to be addressed. In Step 6 the flagged guideline violations and consistency issues are examined and handled accordingly. This is an activity of the design process and is intrinsically manual. It requires that the engineers review the outputs of the previous steps and determine the reported violations and consistency issues that will be handled and how.

The steps of the *operation* phase can be applied iteratively, until the transition criteria from the requirements and design processes to the subsequent development activities (*e.g.*, design, coding, develop verification cases) have been met. Furthermore, the resulting mapping model from Step 4 can be used along the design models to support the subsequent development activities.

3.2 *checsdm4uss*: Concrete Instantiation of *checsdm*

This section describes an instantiation of the *checsdm* approach to one of the design scenarios of industry partners; in essence: avionics systems represented using a mix of UML, Simulink and Stateflow design models. We name this instantiation *checsdm4uss*[2] (*checsdm* for UML, Simulink and Stateflow). This section is organized closely following the three phases of the *checsdm* approach described in Section 3.1. Subsection 3.2.1 first describes the execution of the elicitation phase. Subsection 3.2.2 goes over the codification of metamodels, model importers, modelling tool connectors, the design guidelines and the mapping rules. Afterwards, Subsection 3.2.3 talks about the operation phase applied to the LGCS running example.

[2] Pronounced "checks them for us".

3.2.1 *checsdm4uss*: Elicitation phase

checsdm4uss is motivated in direct response to the industry partners' needs during their avionics system developments and certification with DO-178C, DO-331 and DO-332. Their design scenario is characterized by employing different combinations of UML, Simulink and Stateflow design models to describe the different aspects of systems they develop. The *elicitation* phase was carried out iteratively. Base sets of mapping rules and design guidelines were initially developed after careful analysis of the modelling languages and the main guidelines for their use (*i.e.* the MAAB style guidelines, the UML specification, DO-178C, DO-331 and DO-332). Overall, the specified mapping rules and design guidelines target the provisioning of evidence for achievement of DO-178C verification objectives regarding design process outputs.

These sets were then refined with input from several practitioners within the partner organizations, and, additionally, from their initial applications during the execution of the *operation* phase over different avionics systems. Nonetheless, the resulting sets of mapping rules and design guidelines are not final. The inherent complexity of developing an heterogeneous design for an embedded control software as well as company-specific practices and the UML, Simulink and Stateflow modelling languages themselves, make it hard to determine all possible mapping rules and design guidelines. The proposed mapping rules and design guidelines cover the most common overlap cases between elements in the design models that, when not dealt with a rigorous and systematic approach, lead to consistency issues. Additional guidelines and mapping rules can be added to expand the range of mapping and guidance.

3.2.1.1 Mix of modelling languages

The elicited mix of UML, Simulink and Stateflow was motivated by the industry partners' particular interest to have a complete design redundancy with these modelling languages. This resulted in the following feature configuration from the feature diagram in Figure 3.2: 1) complete coverage of the system elements, 2) provision of both structural and behavioural perspectives, 3) description of elements at the same level of abstraction, and 4) complete overlap

of elements. Regarding UML, the industry partners use the minimal set of constructs related to classes, components and state machines. For Simulink and Stateflow, the minimal set was made up of Simulink subsystem blocks and Stateflow charts (including the charts' internal constructs).

3.2.1.2 Mapping rules

UML and Simulink are fundamentally different modelling languages, yet syntactical and semantical relations can be provided as mapping rules. Given the subset of constructs used by the industry partners, twenty mapping rules were defined between UML, Simulink and Stateflow. UML state machines and Stateflow share the same semantic domain and many syntax constructs. This close "distance" resulted in the majority of mapping rules being defined between UML state machine and Stateflow constructs. Table 3.1 lists all the mapping rules defined between these modelling languages. Mapping rules were established following a top-down strategy, starting from the high-level constructs (*e.g.*, UML component and Simulink subsystem block) to lower-level ones (*e.g.*, UML input parameter and Simulink block input) in order to properly capture the relationships.

Table 3.2 shows in detail mapping rule mr_us_03 describing the relationship between UML components and Simulink subsystem blocks. In Simulink, subsystem blocks gather blocks for the purpose of model organization and visual simplification, as well as for maximizing design reuse. Simulink subsystem blocks are semantically equivalent to UML components, which are considered modular autonomous units with hidden internals but well-defined interfaces that enable reuse. A UML component and a Simulink subsystem block are related *when* both of these elements have similar names. Similarity between names was defined as having an edit distance of no more than 20 percent[3], otherwise, they should be synonyms. However, for this relationship to hold in its entirety, the inputs, outputs and nested elements of both constructs should also be related. Thus, other mapping rules (*i.e.* mr_us_03, mr_us_05, mr_us_06, mr_-

[3] This edit distance was set arbitrarily.

Table 3.1 Summary of mapping rules for *checsdm4uss*.

ID	Name
mr_us_01	UML class and Simulink subsystem
mr_us_02	UML class and Stateflow chart
mr_us_03	UML component and Simulink subsystem
mr_us_04	UML component and Stateflow chart
mr_us_05	UML input parameter and Simulink block input
mr_us_06	UML output parameter and Simulink block input
mr_us_07	UML input parameter and Simulink block output
mr_us_08	UML output parameter and Simulink block output
mr_us_09	UML state machine and Stateflow chart
mr_us_10	UML composite state and Stateflow composite state
mr_us_11	UML region and Stateflow parallel state
mr_us_12	UML state and Stateflow state
mr_us_13	UML choice pseudostate and Stateflow junction
mr_us_14	UML fork pseudostate and Stateflow composite state
mr_us_15	UML join pseudostate and Stateflow composite state
mr_us_16	UML default transition and Stateflow default transition
mr_us_17	UML transition and Stateflow transition
mr_us_18	UML transition trigger and Stateflow transition trigger
mr_us_19	UML transition guard and Stateflow transition guard
mr_us_20	UML transition actions and Stateflow transition actions

us_07, mr_us_08) further describing such fine-grained relationships are referenced as *where* clauses.

Table 3.2 Mapping rule mr_us_03 for UML components and Simulink subsystems. Extracted from Paz *et al.* (2020).

Mapping Rule (ID: Name)	mr_us_03: UML component and Simulink subsystem
When	The UML component and the Simulink subsystem have similar names.
Where	
(1)	Input parameters of operations in the UML component's provided interface and inputs of the Simulink subsystem block are matched (referenced rule: mr_us_05, see Table-A III-5).
(2)	Output parameters of operations in the UML component's required interface and inputs of the Simulink subsystem block are matched (referenced rule: mr_us_06, see Table-A III-6).
(3)	Input parameters of operations in the UML component's required interface and outputs of the Simulink subsystem block are matched (referenced rule: mr_us_07, see Table-A III-7).
(4)	Output parameters of operations in the UML component's provided interface and outputs of the Simulink subsystem block are matched (referenced rule: mr_us_08, see Table-A III-8).
(5)	Nested components and nested subsystem blocks are matched (referenced rule: mr_us_03).

Simulink blocks receive inputs through inports and then compute them to generate outputs through outports. In UML, the inputs for a component can come from either input parameters

of operations in the component's provided interfaces, or from output parameters in the component's required interfaces. Figure 3.8 illustrates these situations using excerpts from the LGCS design models. In the figure, UML component SequenceController (left of the figure) and Simulink subsystem block SequenceController (right of the figure) are matched by mapping rule mr_us_03. Both elements have related inputs. The position and ASStatus inputs of the UML component SequenceController (left of the figure) are matched, respectively by mapping rules mr_us_05 and mr_us_06, to the Placement and ASStat inputs of the corresponding SequenceController Simulink subsystem block (right of the figure). In the case of the UML component, these two inputs are specified in two separate interfaces, Command and ControlData respectively; position being an input parameter of the component's provided interface and ASStatus being an output parameter of a required interface. Mapping rules mr_us_05 and mr_us_06 capture such a semantic equivalence. Similarly, outputs in a UML component can come from either input parameters of operations in the component's required interfaces, or output parameters of operations in the component's provided interfaces. Mapping rules mr_us_07 and mr_us_08 capture such a semantic equivalence.

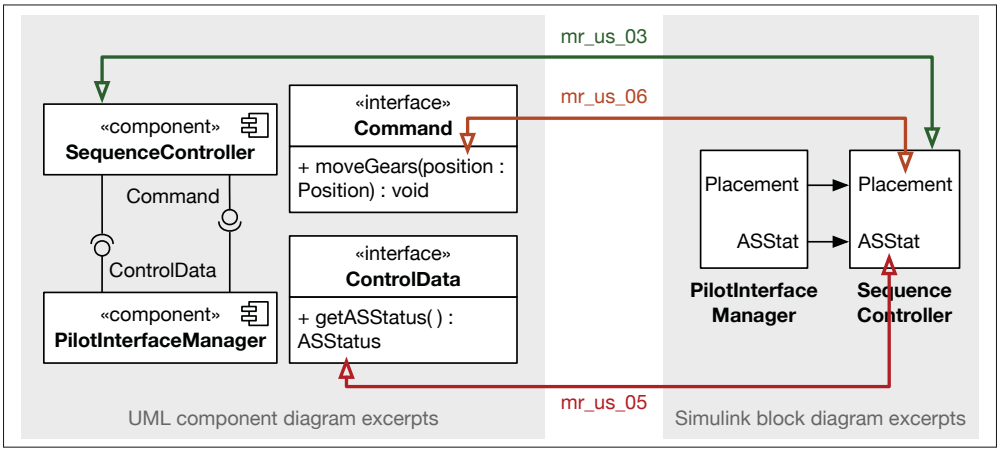


Figure 3.8 Excerpt from the LGCS design models illustrating the application of mapping rules mr_us_03, mr_us_05 and mr_us_06. Extracted from Paz *et al.* (2020).

Table 3.3 shows mapping rule mr_us_05 describing the relationship between UML input parameters and Simulink block inputs. A UML input parameter and a Simulink block input are

related *when* both of these elements have similar names and similar data types. The data type similarity clause was included to help achieve a mandatory DO-332 verification activity. No additional relationships are required to hold, hence, no mapping rules are referenced as *where* clauses.

Table 3.3 Mapping rule mr_us_05 for UML input parameters and Simulink block inputs. Extracted from Paz *et al.* (2020).

Mapping Rule (ID: Name)	mr_us_05: UML input parameter and Simulink block input
When	
(1)	The input parameter's name is similar to the block input's name.
(2)	The input parameter's data type is similar to the block input's data type (referenced DO-332 verification activity: OO.6.2.g).

Note: Empty compartments are omitted to keep the table uncluttered.

All *checsdm4uss*' mapping rules are presented in Appendix III.

3.2.1.3 Design guidelines

Given the subset of constructs used by the industry partners, twenty-one design guidelines (intra- and inter-model) were developed in *checsdm4uss*. Fifteen are set as mandatory and six are recommended. Sixteen are intra-model design guidelines and five are inter-model design guidelines. Table 3.4 lists all the design guidelines. The design guidelines are divided into four categories: 1) suggest an appropriate mix of modelling languages driven by the nature of the system being modelled and the purposes of the models, 2) instruct on the naming of elements in the design models, 3) constrain the use of specific modelling language constructs, and 4) provide semantic equivalences between the constructs of different modelling languages.

Table 3.5 shows design guideline av_us_01 for the mixed use of UML, Simulink and Stateflow in design. The design guideline describes some usage scenarios for these languages based on the nature of the system being modelled. It is not feasible to describe in a single design guideline how to model all possible systems and in a way that is applicable to the design processes of an organization. Thus, engineers must use their knowledge and experience on using the modelling languages to represent portions of the system that better suits the nature

Table 3.4 Summary of design guidelines for *checsdm4uss*.

ID	Type	Title	Priority	Category
av_us_01	Inter-	Mixed use of UML, Simulink and Stateflow	Recommended	1
av_us_02	Intra-	Definition of a naming convention	Mandatory	2
av_us_03	Intra-	Naming of elements in UML models	Mandatory	2
av_us_04	Intra-	Naming of UML fork and join pseudostates	Mandatory	2
av_us_05	Intra-	Naming of elements in Simulink / Stateflow models	Mandatory	2
av_us_06	Intra-	Naming of Simulink inport and outport blocks	Recommended	2
av_us_07	Intra-	Decomposition type for Stateflow chart and composite state	Recommended	3
av_us_08	Intra-	Expression of triggers in UML transitions	Mandatory	3
av_us_09	Intra-	Expression of triggers in Stateflow transitions	Mandatory	3
av_us_10	Inter-	Expression of triggers appearing in both UML and Stateflow transitions	Mandatory	3
av_us_11	Intra-	Expression of UML guards in transitions	Mandatory	3
av_us_12	Intra-	Expression of Stateflow conditions in transitions	Mandatory	3
av_us_13	Intra-	Expression of UML actions	Mandatory	3
av_us_14	Intra-	Expression of Stateflow actions	Mandatory	3
av_us_15	Inter-	Expression of actions appearing in both UML and Stateflow models	Mandatory	3
av_us_16	Intra-	Use of signal receipt and send symbols in UML state machines	Recommended	3
av_us_17	Intra-	Use of UML fork and join pseudostates	Mandatory	3
av_us_18	Intra-	Data type of Simulink inports and outports	Mandatory	3
av_us_19	Intra-	Conjugation of a UML port	Mandatory	3
av_us_20	Inter-	Specification of UML entry and exit points in Stateflow	Recommended	4
av_us_21	Inter-	Specification of UML fork and join behaviour in Stateflow	Recommended	4

of those given portions. Since this guideline has no binding effect on consistency of the design models it is given the recommended priority.

Table 3.6 shows design guideline av_us_03 providing guidance on the naming of elements in UML models. The guideline makes sure the elements in a UML model have a name. It is worth noting that UML modelling tools do not constraint the designer to name the model elements. This guideline is essential for allowing the mapping of design models since finding a match between elements of the design models by means of the defined mapping rules relies heavily on naming.

Table 3.7 shows design guideline av_us_10 providing guidance on the expression of triggers in transitions appearing in both UML state machines and Stateflow charts. The design guideline

Table 3.5 Design guideline av_us_01: Mixed use of UML, Simulink and Stateflow. Extracted from Paz *et al.* (2020).

Guideline (ID: Title)	av_us_01: Mixed use of UML, Simulink and Stateflow
Priority	Recommended
Scope	UML, Simulink and Stateflow
Prerequisites	None
Description	
	The choice of using UML, Simulink or Stateflow, or a mix of them to model all the system or given portions of it should be driven by the nature of the system being modeled and the purposes of the models.
•	If the system (or the portions of it) primarily involves software, use UML components and classes to characterize the structure and behaviour of the software.
•	If the system (or the portions of it) involves both software and hardware elements, use UML components and classes to characterize the structure and behaviour of the software, and use Simulink and Stateflow to characterize the structure and behaviour of the hardware elements.
•	If the behaviour of the system (or the portions of it) primarily involves modal logic with a combination of past and present logical conditions, use UML state machines, Stateflow charts or a mix of them.
•	If a behaviour segment primarily involves behaviour that may execute concurrently, use UML state machines.
•	If the behaviour of the system (or the portions of it) primarily involves if-then-else statements, use Stateflow truth table charts.

Note: Some compartments are omitted to keep the table uncluttered.

Table 3.6 Design guideline av_us_03: Naming of elements in UML models. Extracted from Paz *et al.* (2020).

Guideline (ID: Title)	av_us_03: Naming of elements in UML models
Priority	Mandatory
Scope	UML
Prerequisites	None
Description	
	All `NamedElements` in a UML model must have a name.

Note: Some compartments are omitted to keep the table uncluttered.

determines the notation and contents of a trigger of a transition in both cases. The design guideline reuses existing knowledge from the study in (Ferrari *et al.*, 2013) for dealing with verification issues when choosing to use events in Stateflow transitions and advises otherwise. Figure 3.9 illustrates the application of this design guideline in the context of the LGCS design models. The guideline states that if a transition appears in a UML state machine (left of the figure) as well as a Stateflow chart (right of the figure), then the Stateflow transition must

be guarded by a condition variable whose name must be a substring of the UML transition's trigger event name. An exception applies with regards to relative time event triggers on both languages, which are denoted with "`after`" followed by a number and unit of time. This design guideline is essential for allowing a mapping, by means of the mapping rules, to be established between triggers in UML state machines and triggers in Stateflow charts. Thus, guideline av_us_10 is given the mandatory priority.

Table 3.7 Design guideline av_us_10: Expression of triggers appearing in both UML and Stateflow transitions. Extracted from Paz *et al.* (2020).

Guideline (ID: Title)	av_us_10: Expression of triggers appearing in both UML and Stateflow transitions
Priority	Mandatory
Scope	UML and Stateflow
Prerequisites	
•	Guideline av_us_08: Expression of triggers in UML transitions
•	Guideline av_us_09: Expression of triggers in Stateflow transitions
Description	
	If a transition appears in a UML state machine as well as a Stateflow chart, then the name of the Stateflow condition variable must be a substring of the UML trigger event name. An exception to the previous restriction applies to relative time event triggers, which must be denoted with "`after`" followed by the argument values corresponding to the number and unit of time.

Note: Some compartments are omitted to keep the table uncluttered.

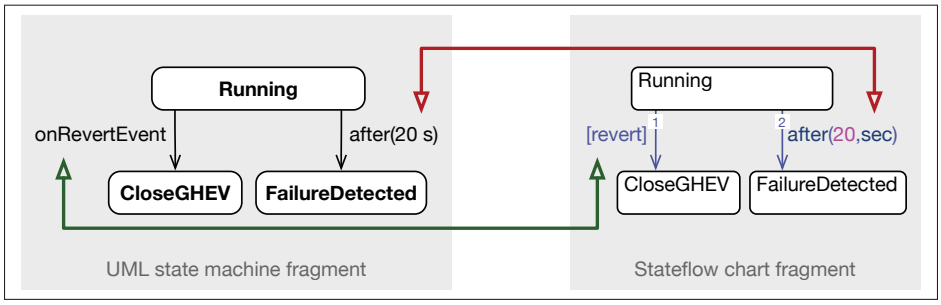


Figure 3.9 Illustrative example for applying guideline av_us_10. Extracted from Paz *et al.* (2020).

Design guideline av_us_18 providing guidance on the data types of Simulink Inports and Outports is also essential for allowing mappings, by means of the mapping rules, to be established

between inputs and outputs of UML components and Simulink subsystem blocks. This guideline was derived from a mandatory DO-332 data type consistency verification activity.

All *checsdm4uss*' design guidelines are presented in Appendix III.

3.2.2 *checsdm4uss*: Codification phase

3.2.2.1 Metamodelling

Creation of UML models is supported by the Eclipse IDE and its EMF-based UML plug-in. The Eclipse Papyrus plug-in (The Eclipse Foundation, 2017b) was integrated as well to provide graphical facilities to visualize and edit the UML models. On the contrary, development of Simulink and Stateflow models is restricted to MATLAB. Simulink and Stateflow are proprietary modelling languages of MathWorks limited to the MATLAB environment. A specification of these languages is also not openly available.

A number of tools have been developed to support Simulink and Eclipse integration (*e.g.*, Lyo OSLC Simulink Adapter (Eclipse Lyo, 2014), Massif (IncQuery Labs, 2017)). Others like Matclipse (Camhy *et al.*, 2013) have focused only on the MATLAB Workbench integration without providing support for Simulink and Stateflow. Furthermore, most of them are no longer maintained and, thus, do not work with the most recent releases of the Eclipse platform and its MDE tools. Massif is the most actively maintained tool. It is a live bridge between Simulink and Eclipse for enabling MDE workflows. Its main feature in the context of *checsdm* is that it provides an EMF-based Simulink metamodel. However, this metamodel lacks support for Stateflow and certain implementation choices reduce its interoperability with other Eclipse tools for MDE. The metamodel has been enriched with (Viatra) query-based derived attributes that, as claimed by Horváth *et al.* (2015), facilitate both understanding and working with complex Simulink models. Viatra queries in an EMF metamodel pose an issue when using a resulting model in other EMF-based Eclipse tools, like Epsilon. Such model queries are not executed, hence, it results in missing information and interoperability problems. Despite

this issue, Massif's EMF-based Simulink metamodel was taken as the basis for this metamodelling step of *checsdm*'s execution. *checsdm4uss*' Simulink and Stateflow metamodel was built from scratch, using EMF, based on Massif's metamodel for the Simulink constructs and adding new constructs corresponding to Stateflow. Development of the metamodel was iterative and involved practitioners from industry who validated several design models conforming to it.

Processing the contents from Simulink and Stateflow models requires their transformation from the MathWorks format into the developed EMF-based metamodel. Therefore, we developed *Breesse*, a bridge for the EMF ecosystem and the MathWorks Simulink and Stateflow ecosystem. *Breesse* is capable of 1) extracting the information from the Simulink and Stateflow models opened in MATLAB, and 2) creating new EMF models conforming to the developed Simulink and Stateflow metamodel. The complete description of *Breesse* is available in Appendix V.

3.2.2.2 Codification of mapping rules

Recall from Section 3.1.2, that mapping rules are codified using ECL; each mapping rule being codified as an ECL`MatchRule`. Listing 3.1 shows the codification for mapping rule mr_us_03 (see Table 3.2) as a `MatchRule`. Mapping rule mr_us_03 describes the relationship between UML components and Simulink subsystem blocks (see lines 2 and 3 of the listing). Instances of these elements are mapped *when* they have similar names (see line 5 of the listing). Similarity in element names as defined for the *when* clauses of the mapping rules is given by a hybrid strategy. The current codification of the `matches` operation for string types (see line 5 of the listing) applies first the Jaro–Winkler distance metric to determine their similarity and rule out word contractions. If the metric equates to a similarity of less than 80 percent, then the WordNet (Miller, 1995) lexical database is queried using MIT's Java WordNet interface (JWI) (Finlayson, 2014). Mapping rules in the *where* clauses of mapping rule mr_us_03 are indirectly referenced through a call to ECL's built-in `matches` operation (see lines 8, 11 and 14). For instance, *where* clause 5 states that for the mapping to hold, nested elements from the mapped elements must also be mapped. Line 12 of the listing, queries the UML component

Listing 3.1: Codification of mapping rule mr_us_03. Extracted from Paz *et al.* (2020).

```
1  rule MatchUMLComponentAndSimulinkSubsystemBlock
2  match component : UML!Component
3  with subsystem : Simulink!SubSystem {
4   compare {
5    var sameNames : Boolean = component.name.matches(subsystem.name);
6    var componentInputs = component.obtainInputs();
7    var subsystemInputs = subsystem.obtainInputs();
8    var sameInputs : Boolean = componentInputs.matches(subsystemInputs);
9    var componentOutputs = component.obtainOutputs();
10   var subsystemOutputs = subsystem.obtainOutputs();
11   var sameOutputs : Boolean = componentOutputs.matches(subsystemOutputs);
12   var nestedComponents = component.obtainNestedElements();
13   var nestedSubsystems = subsystem.obtainNestedElements();
14   var sameNestedElements : Boolean = nestedComponents.matches(nestedSubsystems);
15   return sameNames and sameInputs and sameOutputs and sameNestedElements;
16  }
17 }
```

for all its nested components. Line 13 of the listing does the same for the Simulink subsystem block. In line 14 of the listing, ECL applies mapping rule mr_us_03 to the cross product of the resulting collections from lines 12 and 13. Codification of other mapping rules is analogous.

Once all the mapping rules are executed, as stated in the description of *checsdm*, we need to cleanup the mapping model. In *checsdm4uss*, we chose a very conservative strategy for the mapping model clean up heuristics. The mapping model clean up heuristic consists on removing only mappings with inconsistency flags where one of the involved elements was already successfully matched (*i.e.* flagged as consistent) in another mapping. Thus, not all consistency issues identified through the mappings in the resulting mapping model will indeed be consistency issues and could be intentional. Since the context of usage for *checsdm4uss* is safety-critical avionics systems, the preference from the industry partners was to retrieve all inconsistencies even if the side-effect was the reporting of false inconsistencies. They considered reviewing all the mappings to be worthwhile.

3.2.2.3 Codification of design guidelines

Recall from Section 3.1.2, that design guidelines are codified using VQL; each design guideline being codified as a set of VQL queries. Listing 3.2 presents the codification of intra-model design guideline av_us_03 (see Table 3.6) as two model queries. The query `umlNamedEle-`

Listing 3.2: Codification of intra-model design guideline av_us_03. Extracted from Paz *et al.* (2020).

```
1  @Constraint(severity = "error", key = {namedElement},
2   message = "The element must have a name.",
3   targetEditorId = "org.eclipse.uml2.uml.editor.presentation.UMLEditorID")
4  pattern umlNamedElementEmptyName(namedElement : NamedElement) {
5   neg find umlNamedElementName(namedElement);
6  }
7  pattern umlNamedElementName(namedElement : NamedElement) {
8   NamedElement.name(namedElement, name);
9   check (name.matches(".+"));
10 }
```

mentEmptyName (lines 4 through 6) captures the erroneous situation where an element in a UML model does not have a name. This is accomplished by the negative composition of another graph pattern umlNamedElementName (lines 7 through 10), which captures elements in the UML model that have a name. @Constraint annotations, like the one in lines 1 through 3, specify to the Viatra framework that a GraphPattern represents a well-formedness constraint linked to the UML Eclipse editor (see line 3).

Listing 3.3 shows the codification of inter-model design guideline av_us_10 (see Table 3.7) providing guidance on the expression of triggers of a transition appearing in both UML state machines and Stateflow charts. The query matchingTriggerInvalid states that if a transition appears in a UML state machine as well as a Stateflow chart (lines 4 through 6), then the Stateflow transition must be guarded by a condition variable whose name must be a substring of the UML transition's trigger event name (line 10). An exception to the previous restriction applies with regards to relative time event triggers on both languages, which must be denoted with "after" followed by the argument values corresponding to the number and unit of time (line 10). The defined GraphPatterns for these design guidelines are also given the @Constraint annotation (lines 1 through 3). Facilities to initiate the validation and display markers on the instance models are automatically generated when the Viatra Validation Framework processes the annotations. Codification of other design guidelines is analogous.

Listing 3.3: Codification of inter-model design guideline av_us_10. Extracted from Paz *et al.* (2020).

```
1  @Constraint(severity = "error", key = {mapping},
2   message = "The Stateflow trigger must be a substring of the event name of
        $umlTrigger$.",
3   targetEditorId = "ca.ets.avio604.mappings.presentation.MappingsEditorID")
4  pattern matchingTriggerInvalid(mapping : Mapping, umlTrigger : Trigger, sfwTrigger :
        SFWTrigger) {
5   Mapping.left(mapping, umlTrigger);
6   Mapping.right(mapping, sfwTrigger);
7   Trigger.event(umlTrigger, event);
8   Event.name(event, umlEventName);
9   SFWTrigger.statement(sfwTrigger, sfwTriggerStatement);
10  check(!umlEventName.contains(sfwTriggerStatement.substring(1, sfwTriggerStatement.
        length() - 1)) && !(umlEventName.startsWith("after") && sfwTriggerStatement.
        startsWith("after")));
11 }
```

3.2.2.4 Derived toolchain

An Eclipse toolchain was derived with the previous codifications to support the *operation* phase of *checsdm4uss*. Figure 3.10 depicts an overview of the derived toolchain. Eclipse plug-ins contributing contextual menu options to map two model files in distinct languages selected from the IDE's Explorer View have been developed. This menu option provides users with a simplified way of initiating the mapping of the design models. A screenshot of the *checsdm4uss* derived toolchain is presented later with the LGCS running example in Section 3.2.3. Developer's and user's guides for *checsdm4uss* are available in Appendix IV and online (Paz & El Boussaidi, 2019f).

3.2.3 *checsdm4uss*: Operation phase

This subsection presents the results of executing the operation phase of *checsdm4uss* over the LGCS example described in Chapter 2.

3.2.3.1 Step 1: Software specification

This step is described and discussed in the following chapter (see Chapter 4). As stated before, in this chapter we focus on the design and its verification.

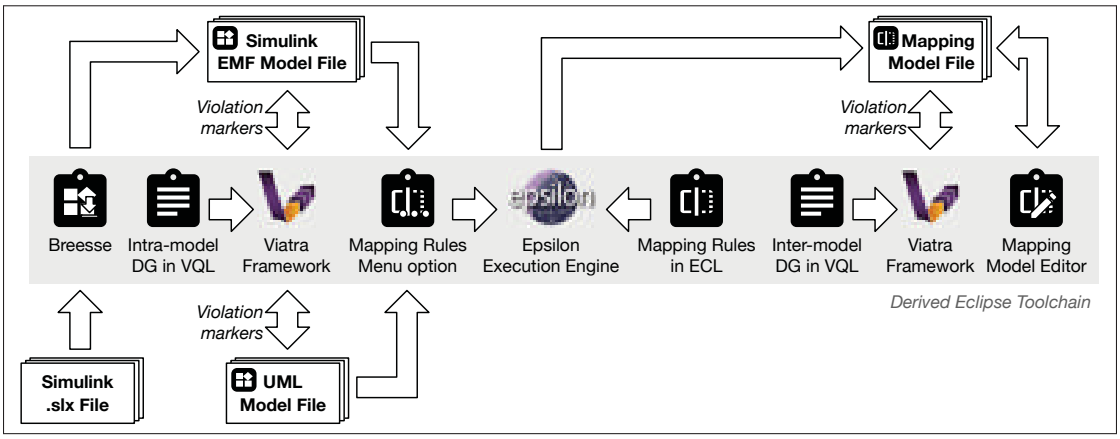


Figure 3.10 Overview of the derived toolchain for *checsdm4uss*.
Extracted from Paz *et al.* (2020).

3.2.3.2 Step 2: Software design

We followed the design procedure described in Section 2.2.3. Some of the design models were presented in Section 2.4. Most of the *checsdm4uss* design guidelines were followed. This was deliberate in order to insert some violations. Following steps of the operation phase give an account of design guideline compliance.

3.2.3.3 Step 3: Verification of intra-model design guideline compliance

The design models in Section 2.4 of the LGS were automatically verified for intra-model design guideline compliance. This was carried out using the *checsdm4uss* toolchain, specifically the design guidelines' codifications as Viatra queries and the generated facilities by the Viatra Validation Framework. The design activity resulted in no deviations from the intra-model design guidelines. An example of a guideline violation is shown later in Step 5.

3.2.3.4 Step 4: Mapping of design models

After the verification of intra-model design guideline compliance, the design models were automatically analyzed in search for correspondences and verified for consistency. This was carried

out using the mapping rules codified with ECL and the support tools developed to initiate the mapping of design models. For instance, mapping rule mr_us_03 presented in Table 3.2 applies to the UML component diagram in Figure 2.9 and the Simulink block diagrams in Figures 2.11 and 2.13. Figure 3.11 displays a screenshot of the *checsdm4uss* mapping tool in action. The left side shows the contextual menu option from which the mapping rules can be invoked over two design models. The right side shows an excerpt of the resulting mapping model from applying the proposed *checsdm4uss* mapping rules to the LGCS UML component diagram and Simulink block diagram. Each element in the model represents a `Mapping`. The bottom of the right side shows an example of the properties in every `Mapping`. The `left` property refers to an element in the UML model and the `right` property refers to an element in the Simulink model.

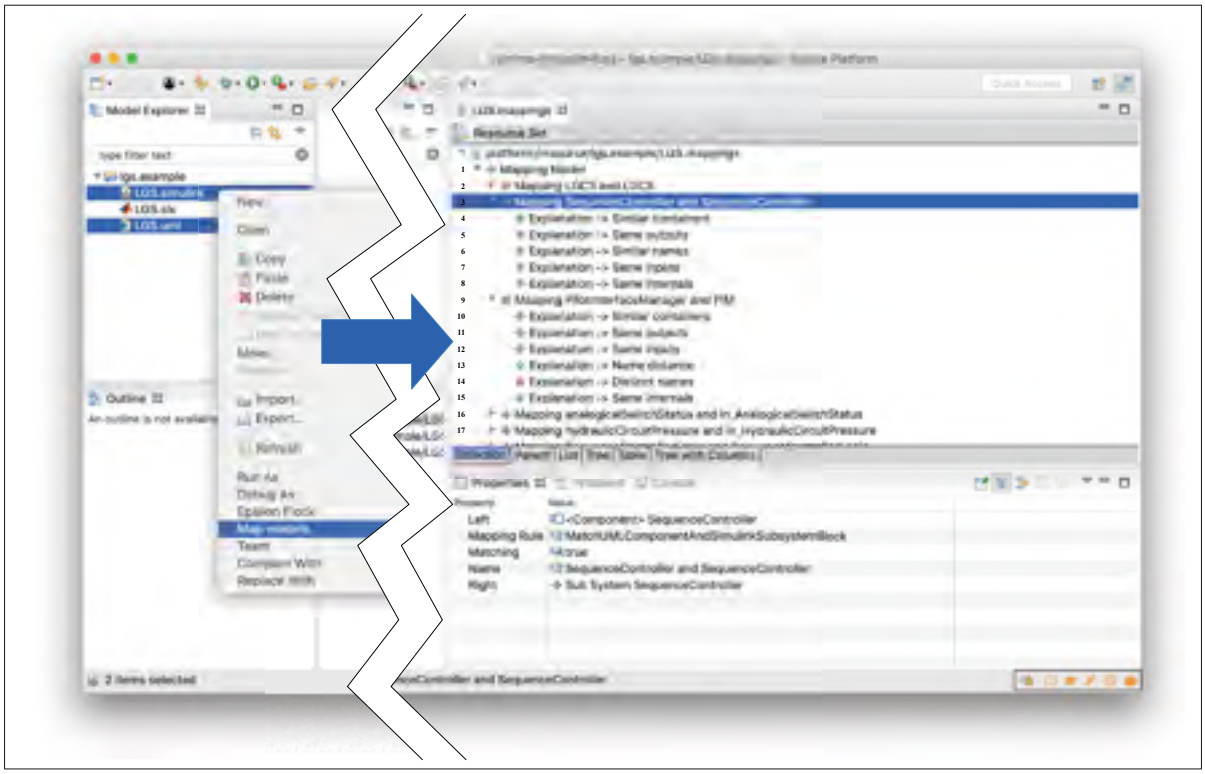


Figure 3.11 Screenshot of the resulting *checsdm4uss* mapping model for the LGCS and the properties of the selected mapping in line 3. Extracted from Paz *et al.* (2020).

Mapping rule mr_us_03 matches the SequenceController component in UML to the SequenceController subsystem block in Simulink (see line 3 of Figure 3.11 right side). The mappingRule attribute informs this as can be seen in Figure 3.11. Both the UML component and the Simulink subsystem block have identical names (see line 6 of Figure 3.11 right side). The other conditions defined in mapping rule mr_us_03 are also met, *i.e.* inputs, outputs and nested elements were matched as well. Lines 5, 7 and 8 of Figure 3.11 (right side) show these results. Lines 16 and 17 of Figure 3.11 (right side) show that the analogicalSwitchStatus and hydraulicCircuitPressure return parameters expected from requiring the ControlData interface (Figure 2.10) are matched to the inports In_AnalogicalSwitchStatus and In_HydraulicCircuitPressure (Figure 2.13), respectively.

The Simulink subsystem block PilotInterfaceManager in Figure 3.11 (right side) was renamed to PIM (see line 9 of Figure 3.11) to deliberately insert a consistency issue. After applying all the mapping rules, the PilotInterfaceManager component and the PIM subsystem block were mapped but not matched (shown in the figure with an icon of a white 'x' enclosed in a red circle). The elements have the same inputs, outputs and nested elements as reported in the explanations in lines 11, 12 and 15. However, they possess distinct names, reason that is disclosed in the explanation in line 14. Since the design scenario involves a complete coverage of system elements and a complete overlapping between the UML and Simulink models, then it is expected that all elements of the UML model are related to a corresponding element of the Simulink/Stateflow model. Thus, the PilotInterfaceManager component and the PIM subsystem block are flagged as inconsistent. This caused an inconsistency propagation to the mapping between the LGCS UML component and the LGCS Simulink subsystem block. This higher-level mapping is, thus, flagged as inconsistent as well (see line 2 of Figure 3.11).

3.2.3.5 Step 5: Verification of inter-model design guideline compliance

The resulting mapping models from Step 4 were used to automatically verify inter-model design guideline compliance. In the *checsdm4uss* instantiation, design guideline av_us_10 (see Table 3.7) was not followed in order to deliberately insert a design guideline violation. In the

UML state machine on the left side of Figure 3.12, the trigger on the transition from the Running state to the close GHEV exit point is expressed as an event name (*i.e.* onRevertEvent). The equivalent transition in the Stateflow chart on the right side of Figure 3.12 is expressed as a condition on an input Boolean variable cancel instead of *revert*. Figure 3.13 displays the error obtained after guideline compliance was verified over the mapping model. Note that in the Stateflow chart on the right side of Figure 3.12, the relative time event trigger after(2,sec) on the transition from the Running state to the FailureDetectedExit state is an example of the exception to the restriction of event names for triggers in Stateflow transitions allowed by design guideline av_us_10.

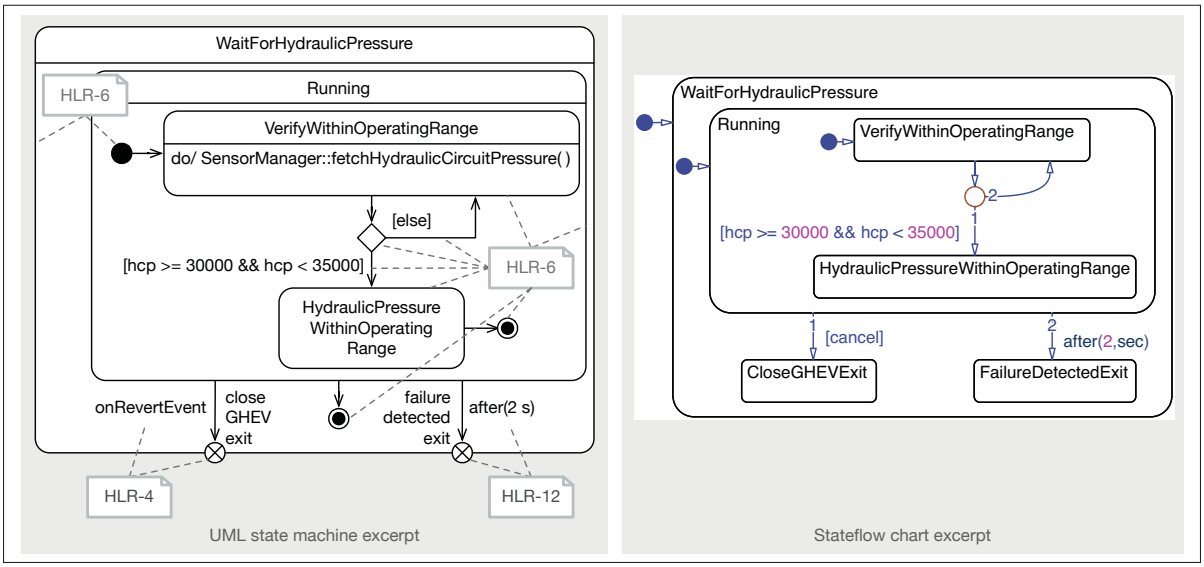


Figure 3.12 Excerpts from the UML state machine associated to the SequenceController component and the Stateflow chart associated to the SequenceController subsystem block. Adapted from Figures 2.12 and 2.14.

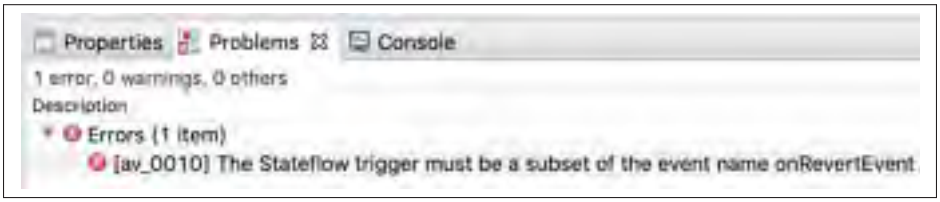


Figure 3.13 Screenshot of design guideline av_us_10's violation. Extracted from Paz *et al.* (2020).

3.3 Chapter Summary

This chapter presented *checsdm*[4] (Consistency of Heterogeneous Embedded Control System Design Models). It is a systematic approach, based on MDE, for assisting engineering teams in ensuring consistency of heterogeneous design of safety-critical avionics software and supporting evidence-gathering efforts for certification. Three features of *checsdm* were highlighted. First, its aim to cover more design scenarios than existing related approaches by providing a design-scenario-independent framework to support verification of heterogeneous design models. Second, its facilities easing the verification of model consistency and adherence to design standards. Third, its enforcement of specific aspects of DO-178C compliance needs in an effort to aid the recollection of evidence for certification. The chapter then described a concrete instantiation of *checsdm* targeting one of the design scenarios of the industry partners; in essence: avionics systems represented using a mix of UML, Simulink and Stateflow design models. This instantiation was named *checsdm4uss*[5] (*checsdm* for UML, Simulink and Stateflow). *checsdm4uss* was presented closely following the three phases of the *checsdm* approach. In particular, the operation phase was illustrated with the LGCS running example.

[4] Pronounced "checks them".

[5] Pronounced "checks them for us".

CHAPTER 4

SPECML: REQUIREMENTS SPECIFICATION MODELLING LANGUAGE

This chapter discusses in greater detail the unfolding and automated support of the first step of *checsdm*'s *operation* phase: the software specification. Recall the development of DO-178C-compliant avionics software begins when a set of high-level system requirements and system safety requirements is allocated to software. Such SRATS must be refined and decomposed into high-level software requirements (HLRs) and documented in the Software Requirements Data *data item*. The software specification step of *checsdm*'s *operation* phase encompasses these activities. Our industry partners were interested in facilitating the step through a modelling language that 1) is easy to use, 2) enables requirements-based verification, and 3) supports traceability by integrating with the other tools used for design.

In order to satisfy these needs we developed SpecML, a modelling language that provides a requirements specification infrastructure for DO-178C-compliant software development. The language is designed as a UML profile augmenting SysML requirements by integrating constructs from several existing approaches. Three features in SpecML can be highlighted. First, it enforces required information (*e.g.*, trace data, decomposition of requirements) for achieving objectives and activities defined in DO-178C. Second, it captures requirements in natural language to smooth the way for its adoption in industry. Third, it provides facilities to capture requirements in a structured and semantically-rich formalism to enable requirements-based analyses and testing. The language is open and may be tailored to specific industries and regulatory guidelines and standards.

The avionics industry considers UML, with profiles such as SysML and MARTE, provides better long term sustainability and interoperability over completely custom-built domain-specific languages Le Sergent *et al.* (2016). Zoughbi *et al.* (2011) compiled a list of benefits justifying the use of UML in the avionics industry. Thus, SysML Requirements was selected as the starting point to build upon. Chapter 1 presented a number of approaches allowing the expression of semantically richer requirements than natural language statements. Of these, the

SysML specification acknowledges the property-based requirement (PBR) theory, defined by Micouin (2008), as an improvement to SysML Requirements addressing a formal expression of requirements in SysML and expanding its ability to support requirements-based analyses and testing.

The chapter is organized as follows. Section 4.1 presents the methodology we followed to build SpecML as a UML profile. Section 4.2 explains the domain metamodel behind SpecML. Section 4.3 describes SpecML as a UML profile. Section 4.4 presents a reference implementation for SpecML.

4.1 Methodology for Developing SpecML

As presented in Chapter 1.2, UML supports the development of DSMLs through its profile mechanism offering semantic variation points and special constructs intended for the language's refinement (*e.g.*, Stereotypes). We specialized the methodology by Selic (2007) and the extension made by Panesar-Walawege *et al.* (2013) (see Figure 1.7) to our aims with SpecML. Figure 4.1 illustrates our methodology for developing SpecML. The general flow of the methodology remains unchanged: develop the conceptual model of the domain (or domain metamodel) and map its concepts to elements in the UML metamodel. Following the activities proposed by Panesar-Walawege *et al.* (2013), our domain metamodel resulted from a careful qualitative analysis of DO-178C's contents. We introduced a set of activities that would refine the previous domain metamodel to integrate the concepts from the PBR theory by Micouin (2008). The UML profile and its associated OCL constraints were then created based on the resulting domain metamodel. Mapping the domain metamodel to the UML metamodel goes through all the domain concepts and identifies the most suitable UML metaclass for each one. A domain concept is defined as a stereotype to be applied over a selected UML metaclass. We carry out several iterations over the process to ensure coverage of the DO-178C guidelines and conformance with UML.

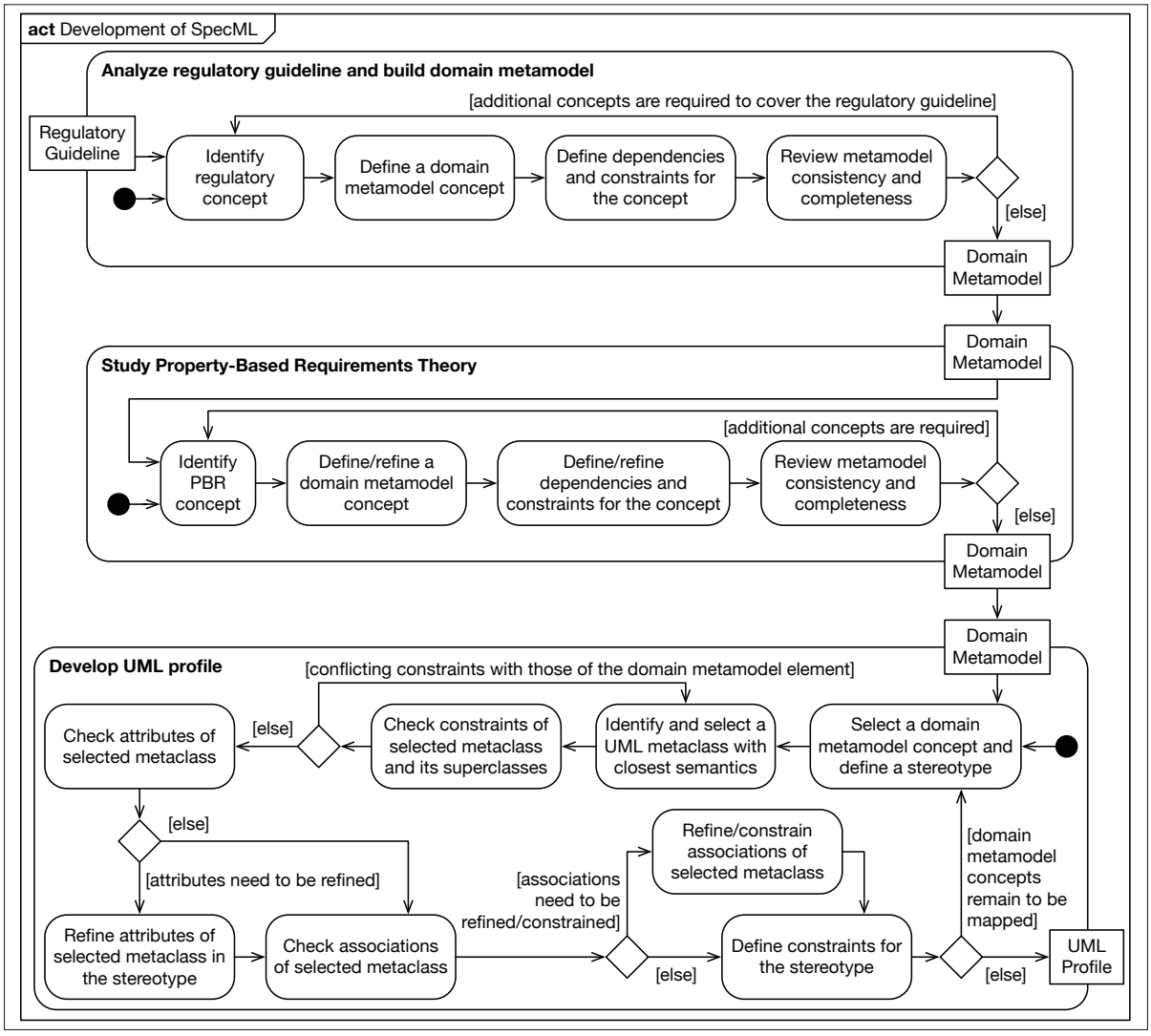


Figure 4.1 Methodology for developing SpecML. Adapted from Figure 1.7.

4.2 SpecML's Domain Metamodel

We use the following template to consistently and uniformly describe the concepts in SpecML's domain metamodel. An example of its use is presented later; however, some attributes and empty template parts are omitted to keep the main thesis document uncluttered. All SpecML's domain metamodel concepts are presented in Appendix VI.

`Concept Name`

Description

Explains the rationale for the existence of the domain concept.

Generalizations

Lists domain concepts that are specialized by the current concept.

Attributes

Lists and describes the characteristic features of the concept. A name and description are provided for each attribute.

Relationships

Lists and describes the relationships of the current concept with other concepts. A name and description are provided for each relationship.

Constraints

Defines the constraints applicable to the concept, its attributes and relationships.

4.2.1 Requirements-related concepts

Figure 4.2 shows a fragment of SpecML's domain metamodel covering the concepts related to requirements as per our analysis of DO-178C. Requirements are a prominent and critical concern for DO-178C-compliant avionics development. The requirements process must develop a prescribed three-level requirements hierarchy. The first level is made up of `SystemRequirements` *allocated to software* (SRATS). SRATS are not specified as part of DO-178C's scope,

they are solely taken as inputs of the DO-178C requirements process. Given SRATS must be *refined* into the second level of requirements, the `HighLevelRequirements` (HLRs) (presented below). A review of the HLRs validates these against the SRATS to check if their intent is being accurately captured by the HLRs in a way suitable for directing software design activities. As mentioned in Chapter 2.5, it was observed during the construction of the LGCS' requirements specification that SRATS can be very detailed, eliminating the need for any further refinement into HLRs. Feedback on the matter from the industry partners pointed out that although such a situation is not infrequent in the industry, eliminating the second level of the requirements hierarchy poses a heavy assurance burden when certifying the software. Hence, the *copy* relationship in the domain metamodel between HLRs and SRATS.

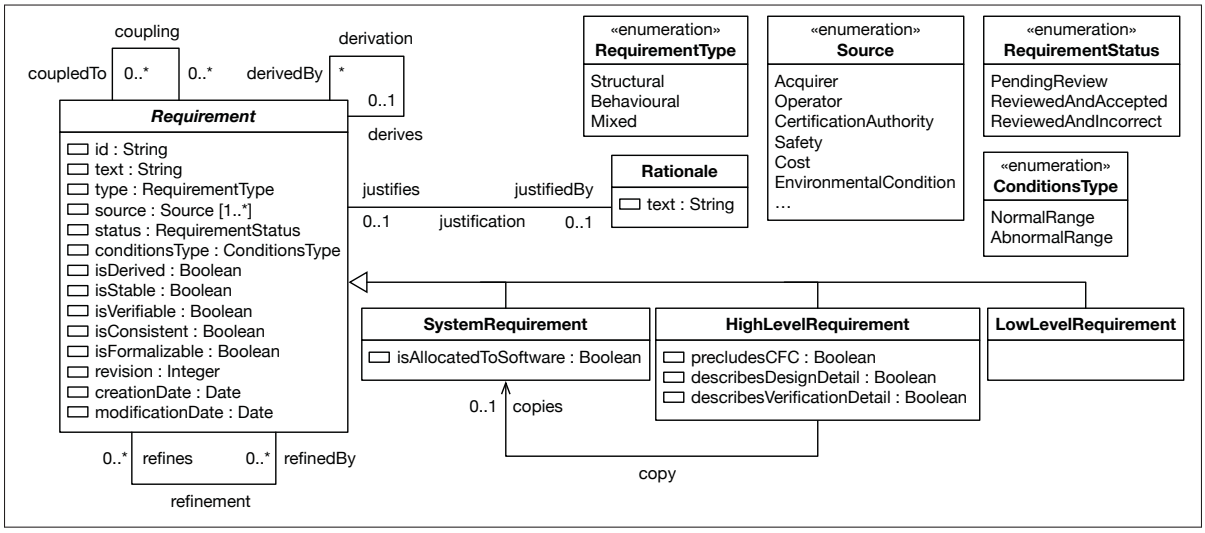


Figure 4.2 Fragment of SpecML's domain metamodel showing the requirements concepts.

`HighLevelRequirement`

Description

A `HighLevelRequirement` (HLR) specifies a capability or condition that must (or should) be satisfied. An HLR is developed from the analysis of SRATS, safety-related requirements and

system architecture. HLRs must be very detailed so as to guide the design activities. HLRs must not include design nor verification details in accordance with DO-178C.

Generalizations

- Requirement

Attributes

- **precludesCFC** : *Boolean [1]*

  Indicates if the requirement intends to prevent one or more of the identified contributions to failure conditions (CFCs).

- **describesDesignDetail** : *Boolean [1]*

  Indicates if the requirement statement describes design detail. HLRs should not describe design details except when there is a justified design constraint.

- **describesVerificationDetail** : *Boolean [1]*

  Indicates if the requirement statement describes verification detail. HLRs should not describe verification details except when there is a justified constraint.

Relationships

- **copy**

  It could occur that allocated system requirements are, in fact, very detailed so as to guide the design without any further refinement into HLRs. In this case HLRs must be defined and related to their corresponding SRATS through the copy relationship. The Copy relationship goes from the HLR to the SRATS.

Constraints

1. A HighLevelRequirement without the isDerived flag must have a refinement or copy relationship to a SystemRequirement.

   This constraint is applicable to ensure traceability to an originating system requirement. Introduced from DO-178C.

2. A HighLevelRequirement with the describesDesignDetail flag must justify the existence of the design detail with a Rationale. Introduced from DO-178C.

   This constraint is applicable to enforce the inclusion of a rationale for DO-178C certification compliance.

3. A HighLevelRequirement with the describesVerificationDetail flag must justify the existence of the verification detail with a Rationale. Introduced from DO-178C.

   This constraint is applicable to enforce the inclusion of a rationale for DO-178C certification compliance.

The correct specification of HLRs initiates the design. The design covers the definition of the software architecture and the refinement of HLRs into LowLevelRequirements (LLRs) (*i.e.* the detailed design). LLRs, together with the architecture, are used to guide the coding and building activities. Although LLRs represent the detailed design, they are still regarded by DO-178C as being requirements and, thus, share many common properties with HLRs. In fact, LLRs can be specified using natural language. We decided to include the LLR concept in our domain metamodel to allow their formalization using SpecML's constructs for such a purpose but also to enable their traceability. Indeed, DO-178C states that HLRs and LLRs must be traced to their originating requirement(s). Whenever a requirement cannot be directly traced to an originating requirement (*e.g.*, because it specifies behaviour beyond that specified by the higher level requirements) it must be identified as a *derived requirement*. Examples of requirements are given with the domain metamodel's mapping to UML stereotypes in Section 4.3.

Furthermore, requirements must exist for both normal-range and abnormal-range (*i.e.* robustness) inputs and conditions. This helps ensure the software will continue to operate correctly to some extent in the face of anomalies. For instance, on the one hand, a normal-range requirement will describe the actuation of a valve when a hydraulic circuit is pressurized to a value within a given range. On the other hand, a robustness requirement will describe the software's behaviour if the pressure value exceeds the maximum defined operational pressure for the hydraulic circuit.

4.2.2 Requirement formalization-related concepts

Figure 4.2 shows a fragment of SpecML's domain metamodel covering the concepts related to requirement formalization as per our analysis of the PBR theory. Recall from Chapter 1 that a PBR for a system Σ is defined as a constraint over a property P of an object O in Σ. The constraint enforces the value of P to be located within a domain D, which is a subset of $im(P)$ (*i.e.* the domain of possible values of P), when a condition C is met. Expression 4.1 formalizes a PBR.

$$\text{Req: } [\text{when } C \rightarrow] \, val(O.P) \in D \subset im(P) \tag{4.1}$$

The `PropertyBasedStatement` concept captures a PBR. We use the word statement to avoid confusion with the different requirement concepts presented above. The conjunction of a finite set of `PropertyBasedStatements` denotes the formalization of a `Requirement`. The term `Req` of Expression 4.1 is a mandatory, unique identifier included as the `id` attribute in the `PropertyBasedStatement` concept. We added the `text` attribute to allow the textual description/reading of the PBR to be captured as well following the template: *"[when condition C is met,] the value(s) of property P of object O shall be in the subset D of* $im(P)$*"*. The presence of a condition C is optional as indicated by the presence of square brackets. This is also evident in the `hasCondition` relationship between the `PropertyBasedStatement` and `Constraint` concepts. The remaining part of the expression represents the predicate. Both the condition and predicate constitute `Constraints` over a set of `Parameters`, regarded as the properties evalu-

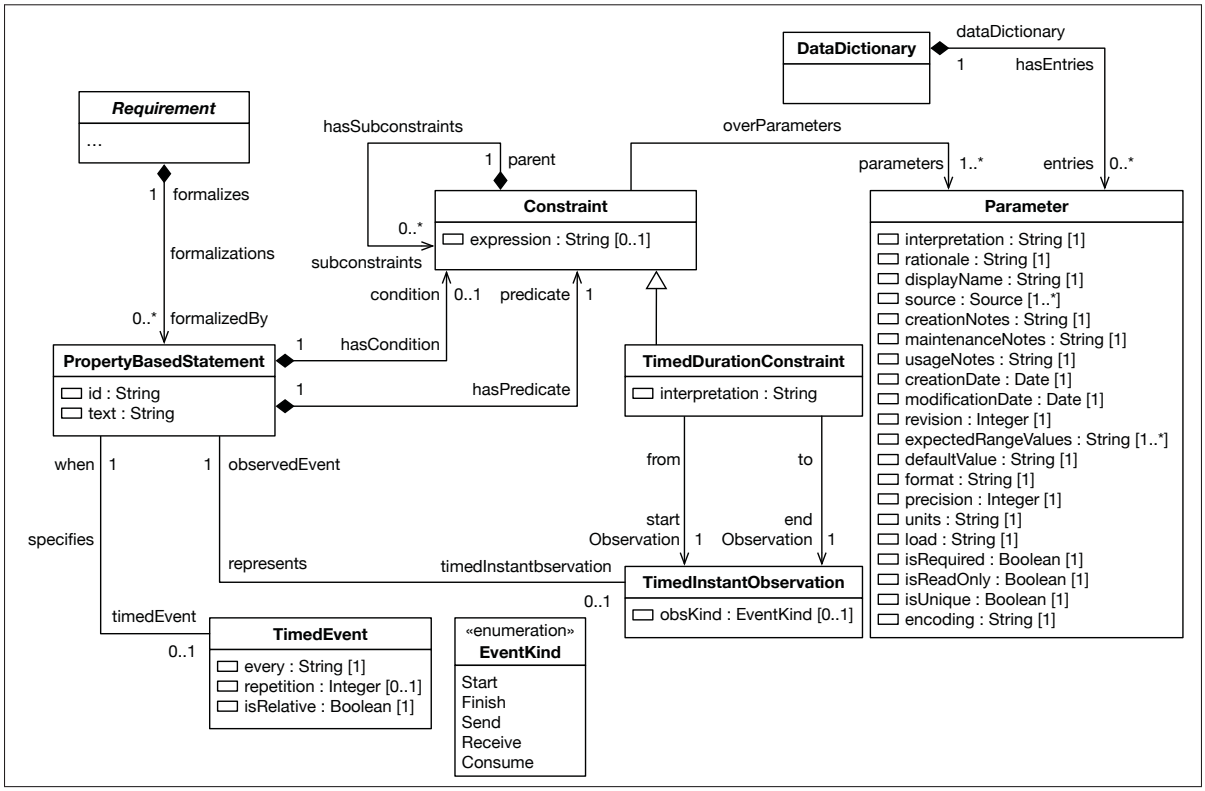


Figure 4.3 Fragment of SpecML's domain metamodel showing the formalization concepts.

ated in a PBR. These Parameters make up the entries in the DataDictionary. The essential metadata for a data dictionary entry were extracted from the DO-178C guideline.

PBR theory lacks constructs for expressing time-dependent behaviour and constraints. In the LGCS, for instance, the behaviour specified in HLR-2 must be carried out in under 28 seconds. Another instance of time-dependent behaviour in avionics software is, for example, when a value of a certain property needs to be evaluated repetitively at a set frequency. This information is part of the PBR but cannot be captured. We added the TimedEvent, TimedDurationConstraint and TimedInstantObservation concepts to the domain metamodel for the expression of such. The TimedEvent concept allows the specification of an annotation to a PropertyBasedStatement indicating the specified behaviour needs to be performed with a predetermined frequency for a number of times. The TimedDurationConstraint concept serves to impose a constraint on the temporal distance between two observed events, repre-

sented through the TimedInstantObservation concept. The TimedInstantObservation concept allows a PropertyBasedStatement to be annotated as an instant in time that will be observed for an event occurrence. Examples of requirement formalizations are given with the domain metamodel's mapping to UML stereotypes in Section 4.3.

4.3 SpecML as a UML Profile

4.3.1 Profile organization

SpecML is designed as a UML profile that augments SysML Requirements with new constructs. SysML 1.5 was selected since this version is more open to extensions regarding requirement specification compared to previous versions. SpecML has the common structure of a UML profile: data types, stereotypes and OCL constraints. No custom data types are defined, hence, UML primitive types are used to define the attributes owned by the stereotypes. Specialized stereotypes are defined to capture the concepts of the domain metamodel described in Section 4.2. OCL constraints are defined to perform the necessary checks regarding the required information for achieving objectives and activities defined in DO-178C. The stereotypes and OCL constraints in SpecML are divided into four groups: 1) requirement hierarchy, 2) requirement interrelationship, 3) requirement formalization, and 4) data dictionary. The following subsection describes in detail the stereotypes and constraints belonging to these groups.

4.3.2 Profile stereotypes

The stereotypes and constraints are documented following a similar layout to the one used in the SysML specification (OMG, 2017a). For each stereotype we use the following documentation template. An example of its use is presented later; however, some attributes and empty template parts are omitted to keep the main thesis document uncluttered. All SpecML's stereotypes are presented in Appendix VII.

`Stereotype Name`

Description

Explains the rationale for the existence of the stereotype and its usage.

Extensions

Lists UML metaclasses extended by the current stereotype.

Generalizations

Lists stereotypes that are specialized by the current stereotype.

Attributes

Lists and describes the characteristic features of the stereotype.

Constraints

Defines the constraints applicable to the stereotype and its attributes.

4.3.2.1 Requirement hierarchy

Figure 4.4 shows the stereotypes to represent the requirement hierarchy concepts of the domain metamodel. The abstract stereotype `Requirement` (presented below) extends the SysML `AbstractRequirement` stereotype. This inheritance provides the facilities to capture natural language requirement statements (the `text` attribute) and specify an identifier (the `id` attribute). On top of that, the SpecML `Requirement` stereotype adds attributes to further characterize a requirement, like `type` (structural, behavioural, mixed), `source` (*e.g.*, acquirer, operator, certification authority, certification standard) and `status` (pending review, reviewed and accepted,

reviewed and incorrect). The isDerived attribute is used to indicate that the requirement is not directly traceable to higher level requirements because it specifies behaviour beyond what has been specified in them.

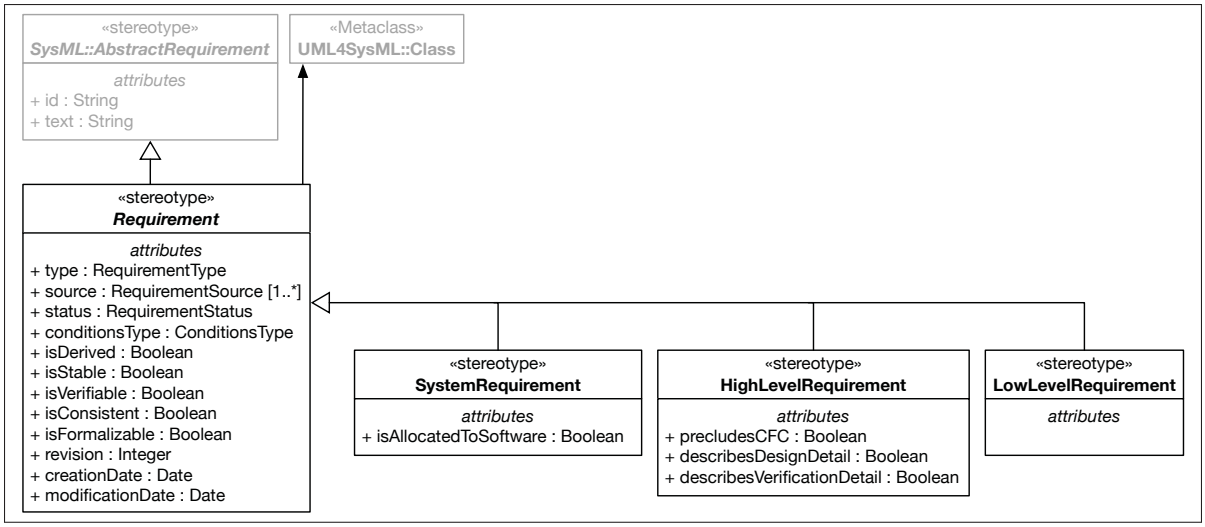


Figure 4.4 Requirement hierarchy stereotypes.

Requirement

Description

A Requirement defines the general attributes and relationships essential for requirements specification of avionics software. Specifically the stereotype generalizes the different kinds of requirements described in DO-178C, namely system requirements, high-level software requirements (HLRs) and low-level software requirements (LLRs). A Requirement conveys an understanding of what needs to be performed by the system of interest.

Extensions

- UML Class

Generalizations

- SysML AbstractRequirement

Attributes

- **id** : *String [1]*

  The unique ID of the requirement. Inherited from `AbstractRequirement` (see clause 16.3.2.1 AbstractRequirement from the OMG (2017a) SysML 1.5 Specification).

- **text** : *String [1]*

  The requirement statement as a natural language statement. Inherited from `AbstractRequirement` (see clause 16.3.2.1 AbstractRequirement from the OMG (2017a) SysML 1.5 Specification).

- **type** : *RequirementType [1]*

  The type of the requirement. Possible values are (Micouin, 2008): structural, behavioural or mixed. Structural requirements concern to a structural property (*i.e.* composition and structure) of the system being specified. Behavioural or functional requirements concern to a behavioural/functional property (*i.e.* observable behaviour) of the system being specified. Mixed requirements concern to both structural and behavioural properties of the system being specified.

- **source** : *Source [1..*]*

  The source who expresses the requirement. Possible values are: acquirer, operator, certification authority, specialty engineer, other stakeholder, or another source like certification standard, safety, costs, environmental conditions, design, production, tests, or maintenance. A requirement may have multiple sources.

- **status** : *RequirementStatus [1]*

The status of the reviewing and acceptance of the requirement. Possible values are: pending review, reviewed and accepted, and reviewed and incorrect.

- **isDerived** : *Boolean [1]*

  Indicates if a requirement is a derived requirement, *i.e.* a requirement that (a) is not directly traceable to a higher level requirement, and/or (b) specifies behaviour beyond that which is specified by the system requirements or the high-level requirements.

Constraints

1. The id must be specified and be unique.

   This constraint is applicable to uniquely identify a requirement.

2. A Requirement with the isDerived flag must be justified by a Rationale (see clause 7.3.2.5 Rationale from the OMG (2017a) SysML 1.5 Specification).

   This constraint is applicable to understand why the requirement cannot be directly traced to an originating requirement because it specifies behaviour beyond that specified by the higher-level requirements.

3. A Requirement, or any of its subclasses, shall not participate as the client in dependencies stereotyped by Satisfy.

   The Satisfy relationship is a dependency between a requirement and a model element that fulfills the requirement. However, when the model element is stereotyped as a LowLevelRequirement the Satisfy relationship takes on the same meaning as the RefineReqt relationship. In such a case the RefineReqt relationship shall be used. Thus, this constraint is intended to enforce the use of the RefineReqt relationship over the Satisfy relationship.

4. A Requirement, or any of its subclasses, shall not own any nested classifiers stereotyped by Requirement.

   This constraint is to avoid the creation of compound requirements and subrequirements.

The SystemRequirement, HighLevelRequirement and LowLevelRequirement stereotypes specialize the Requirement stereotype to define the requirement hierarchy described in DO-178C (see Section 4.2). The SystemRequirement stereotype represents the highest level requirements in the hierarchy. SystemRequirements are system requirements allocated to software (SRATS) when the isAllocatedToSoftware attribute is set to *true*. Recall that SRATS are the ones developed and refined into software requirements. System requirements are specified during the system life cycle processes, beyond the regulatory scope of DO-178C. Only SRATS are considered for software development. However, the SystemRequirement stereotype provides the possibility to formalize any system requirement to enable its analysis and testing as well.

The HighLevelRequirement stereotype (HLR for short, presented in detail below) represents requirements that are produced directly from the refinement of SRATS. Attributes of HighLevelRequirement (*i.e.* precludesCFC, describesDesignDetail and describesVerificationDetail) support analyses for certification and safety compliance. For instance, OCL constraints defined over the last two attributes enforce the specification of a rationale when these attributes are set to true. A rationale is mandatory in these situations for the case of DO-178C certification. The precludesCFC attribute indicates if the requirement intends to prevent one or more of the identified contributions to the system's failure conditions. The LowLevelRequirement stereotype (LLR for short) represents the requirements that can be directly implemented/realized without further information, *i.e.* the detailed design. This stereotype can be used as a stand-alone element to capture natural language or formal requirement statements, or be applied onto UML/SysML model elements that represent the design.

HighLevelRequirement

Description

A HighLevelRequirement (HLR) specifies a capability or condition that must (or should) be satisfied. An HLR is developed from the analysis of SRATS, safety-related requirements and

system architecture. HLRs must be very detailed so as to guide the design activities. It could occur that allocated system requirements are, in fact, very detailed so as to guide the design without any further refinement into HLRs. In this case HLRs must be defined and related to their corresponding SRATS using the Copy relationship. The Copy relationship goes from the HLR to the SRATS. HLRs must not include design details nor include verification details in accordance with regulations.

Generalizations

- Requirement

Attributes

- **precludesCFC** : *Boolean [1]*

  Indicates if the requirement intends to prevent one or more of the identified contributions to failure conditions (CFCs).

- **describesDesignDetail** : *Boolean [1]*

  Indicates if the requirement statement describes design detail. HLRs should not describe design details except when there is a justified design constraint.

- **describesVerificationDetail** : *Boolean [1]*

  Indicates if the requirement statement describes verification detail. HLRs should not describe verification details except when there is a justified constraint.

Constraints

1. A `HighLevelRequirement` stereotype must not be applied alongside other stereotypes that specialize the `Requirement` stereotype.

2. A HighLevelRequirement without the isDerived flag must have a RefineReqt or Copy dependency to a SystemRequirement.

   This constraint is applicable to ensure traceability to an originating system requirement.

3. A HighLevelRequirement with the describesDesignDetail flag must justify the existence of the design detail with a Rationale (see clause 7.3.2.5 Rationale from the OMG (2017a) SysML 1.5 Specification).

   This constraint is applicable to enforce the inclusion of a rationale for certification compliance.

4. A HighLevelRequirement with the describesVerificationDetail flag must justify the existence of the verification detail with a Rationale (see clause 7.3.2.5 Rationale from the OMG (2017a) SysML 1.5 Specification).

   This constraint is applicable to enforce the inclusion of a rationale for certification compliance.

Figure 4.5 presents an SRATS, SRATS-2, from the LGCS and its developing HLR, HLR-2, captured using SpecML's requirement hierarchy stereotypes. The natural language text and IDs of both requirements are captured by the text and id attributes. The requirements describes the behaviour of the LGCS when a desired gear position is given by the pilots, thus, the *BehaviouralRequirement* type. HLR-2 is indicated to preclude CFC-1. Other HLR attributes were set to false.

4.3.2.2 Requirement interrelationship

Figure 4.6 presents the stereotypes for defining the types of relationships that can occur between requirements. The RefineReqt stereotype is an alias to the SysML DeriveReqt stereotype to align it with normative vocabulary. The stereotype represents a bidirectional trace in which a requirement can be refined into a lower-level requirement. This relationship goes from

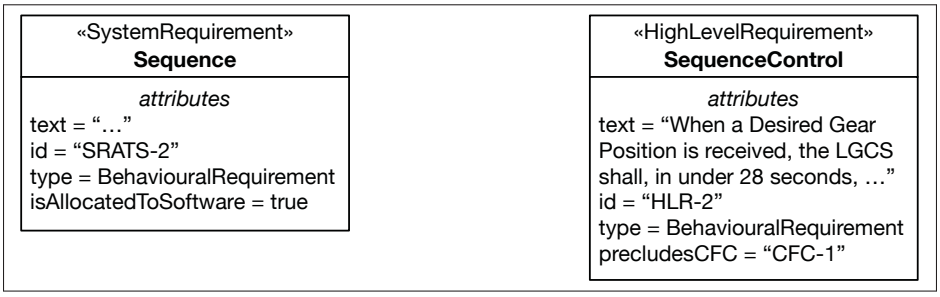


Figure 4.5 Excerpt of the LGCS specification model using SpecML showing SRATS-2 and HLR-2.

the refining requirement (*e.g.*, the HLR) to the refined requirement (*e.g.*, the SRATS). Figure 4.7 presents the use of the `RefineReqt` stereotype on the example from Figure 4.5 to bind SRATS-2 with its developing HLR, HLR-2.

The `Copy` relationship is as defined in SysML but with a more constrained usage. HLRs must be very detailed so as to guide design. However, it could occur that system requirements are, in fact, very detailed so as to guide the design without any further refinement into HLRs. In this case HLRs must be defined and related to their corresponding system requirement using the `Copy` relationship, which will indicate that they are a read-only copy of the supplier requirement (*i.e.* the system requirement). The `Derive` stereotype is included to make it possible to trace derived requirements to the requirements from which they were derived. A derived requirement and the requirement from which it was derived stand at the same hierarchical level. Thus, this relationship enables an indirect trace from the derived requirement to a higher level requirement. Requirements at the same level of the requirements hierarchy may experience some interdependence. The `Coupled` stereotype makes it possible to represent such a relationship between two requirements.

4.3.2.3 Requirement formalization

Figure 4.8 shows the stereotypes to capture requirements in a structured, semantically-rich formalism. The `PropertyBasedStatement` stereotype establishes a formalized statement of a requirement following the domain metamodel from Section 4.2. Expression 4.1 is broken

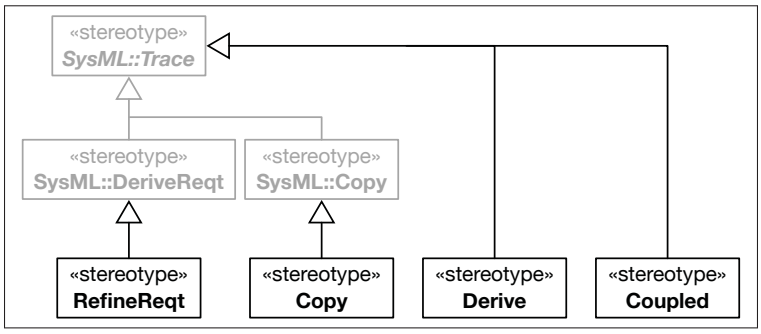


Figure 4.6 Requirement interrelationship stereotypes.

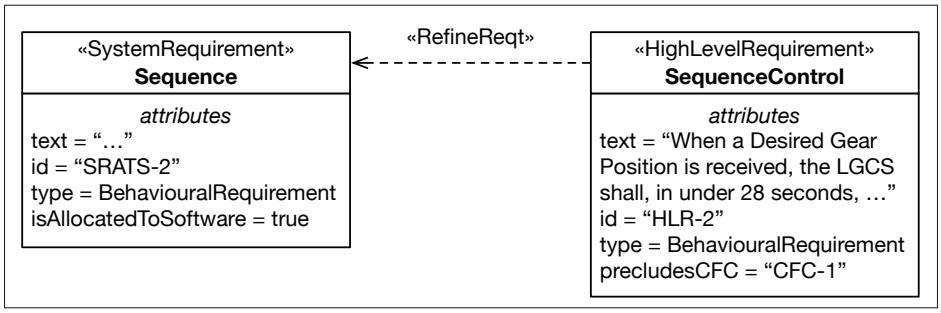


Figure 4.7 Excerpt of the LGCS specification model using SpecML showing the interrelationship between SRATS-2 and HLR-2.

down to simplify its representation with the profile. The `id` attribute in `PropertyBasedStatement` captures the `Req` term of the expression. An additional `text` attribute can hold a textual description of the formalization if necessary. The optional subexpression $[\text{when } C \rightarrow]$ is a *condition* of actualization in the context of the requirement. This expression can be captured with a SysML `ConstraintBlock` and linked to the `PropertyBasedStatement` with a dependency stereotyped by `Condition`. The mandatory subexpression $val(O.P) \in D \subset im(P)$ is a *predicate* representing the constraint over the value of a system property. This expression can be captured as well with a SysML `ConstraintBlock` and linked to the `PropertyBasedStatement` with a dependency stereotyped by `Predicate`. One property-based statement may not be sufficient to capture the entire requirement described in the natural language statement. Thus, additional `PropertyBasedStatements` may be introduced as part of the requirement's

formalization. A dependency stereotyped by Formalization links each PropertyBasedStatement to the requirement. The requirement is, therefore, interpreted as the conjunction of the specified PropertyBasedStatements.

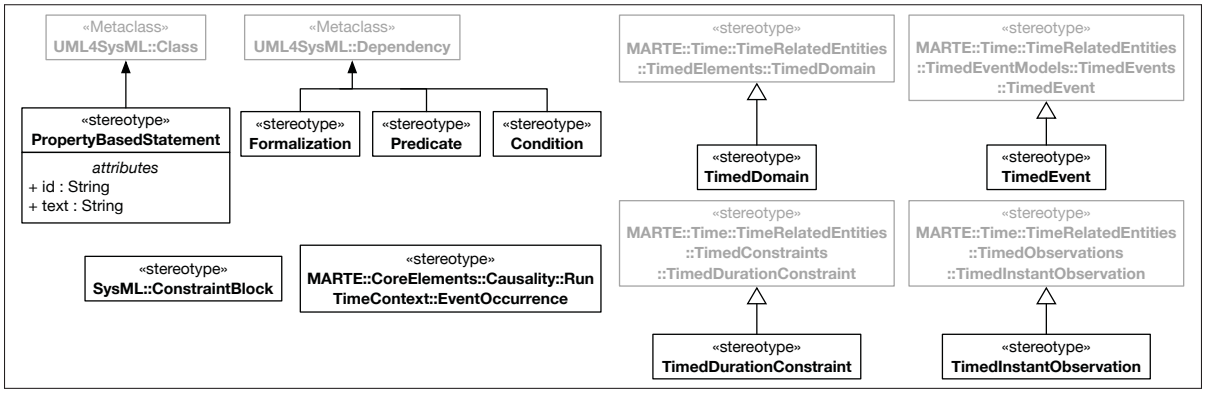


Figure 4.8 Requirement formalization stereotypes.

The LGCS must perform a series of actions as part of HLR-2 and a single formalization statement is not sufficient to cover the whole behaviour. Thirteen property-based statements formalize this HLR. Figure 4.9 shows the first of these formalization statements (2.1). Formalization 2.1 reads as follows: *when the LGCS is functioning normally and is idle, and there is a change in the desired gear position after it has been validated, then the LGCS shall become active*. The condition upon which a gear movement sequence is initiated is captured with StartSequenceControlCondition *ConstraintBlock* and linked to the *PropertyBasedStatement* with the Condition dependency. The system state that must be observed is captured with StartSequenceControlPredicate *ConstraintBlock* and linked to the *PropertyBasedStatement* with the Predicate dependency.

The TimedDomain stereotype indicates a container of clocks and its use is required when working with MARTE. The stereotype must be applied onto a UML Package. Requirements in a TimedDomain package may use the clocks contained in it to express behaviour that is time-dependent. The TimedEvent stereotype is as defined by the MARTE specification but with a more constrained usage. The stereotype establishes a non-functional annotation on a PropertyBasedStatement indicating that the specified behaviour needs to be performed with

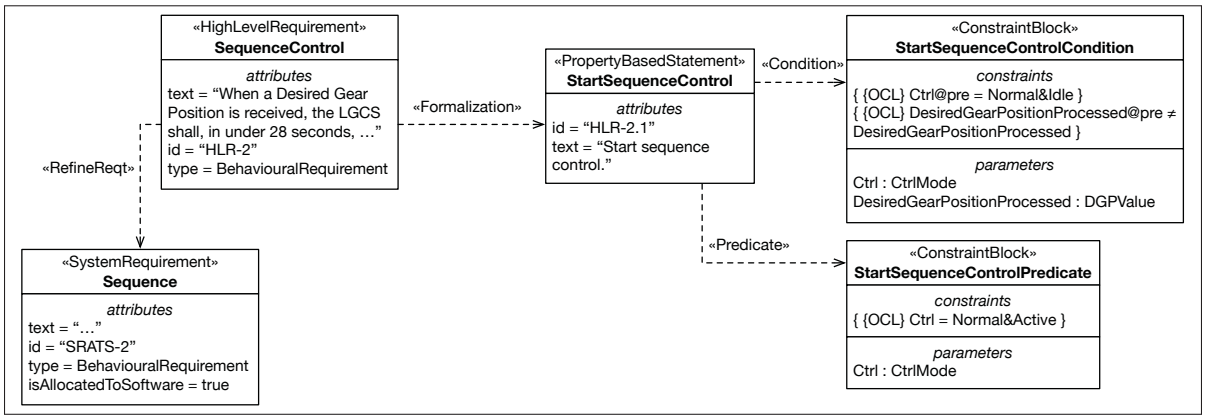


Figure 4.9 Excerpt of the LGCS specification model using SpecML showing one of the formalizations for HLR-2.

a predetermined frequency (*i.e.* it is explicitly bound to a clock). The TimedInstantObservation stereotype is also as defined by the MARTE specification but with a more constrained usage. The stereotype denotes an instant in time that is associated with an event occurrence and observed on a given clock. This stereotype is included to allow the observation of event occurrences and allowing their use in the expression of timing constraints on the specified behaviour. The stereotype must only be applied to a PropertyBasedStatement. The TimedDurationConstraint stereotype imposes a constraint on the temporal distance between two events. This stereotype is included to allow the expression of timing constraints between specified behaviour.

The LGCS' behaviour is time-dependent, for instance, the behaviour expressed in HLR-2 must be performed in under 28 seconds. In Figure 4.10, the package containing HLR-2 and its formalizations is stereotyped by TimedDomain in order to allow the use of SpecML's stereotypes for timing constraints. The figure also displays the HLR-2 formalizations in PropertyBasedStatements 2.1 and 2.13 to be annotated with the TimedInstantObservation stereotype. The annotations can be visualized as comments in Figure 4.10. The instants in time where the LGCS transitions from an idle state into an active state and then back into idle will be observed. These event occurrences represent the start and end of HLR-2's behaviour. A TimedDurationConstraint of 28 seconds is imposed over both TimedInstantObservations.

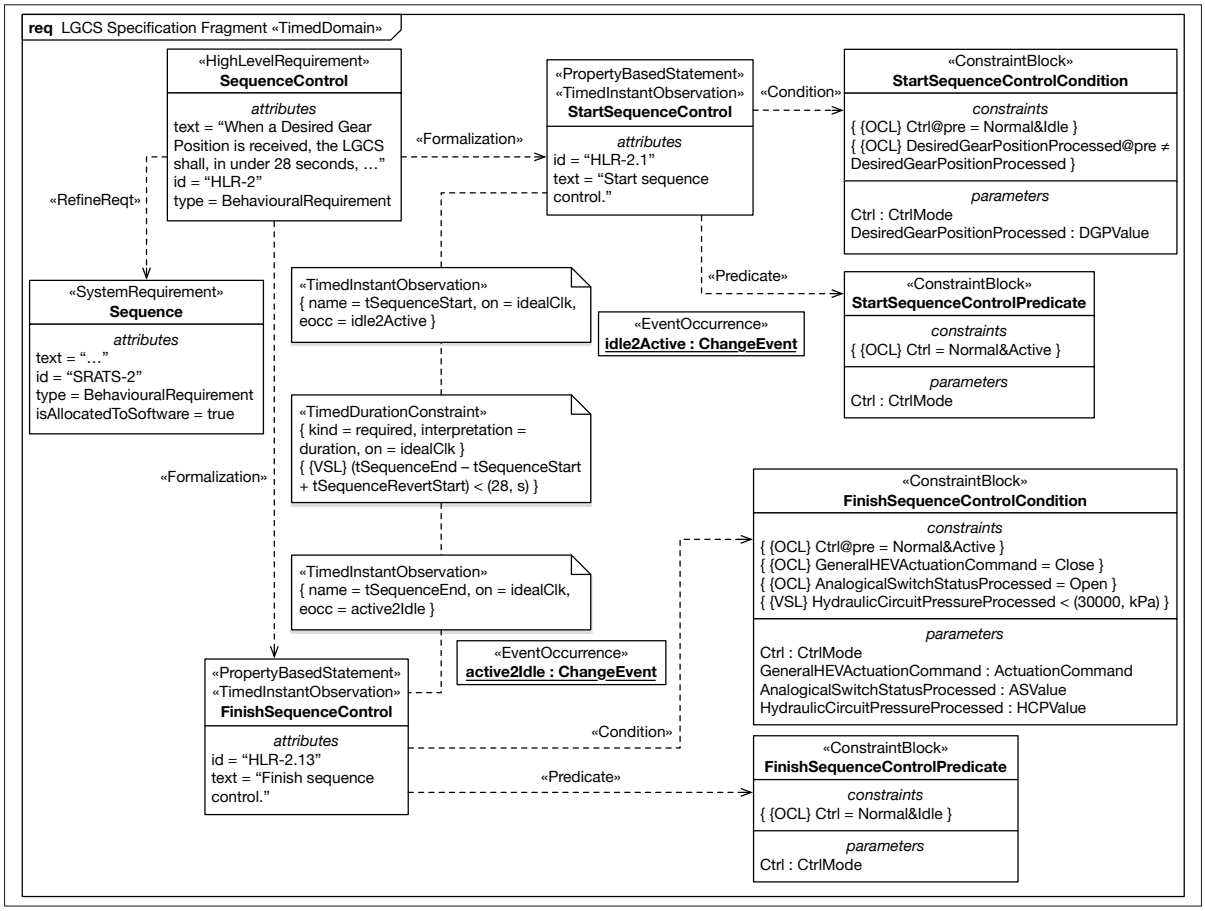


Figure 4.10 Excerpt of the LGCS specification model using SpecML showing timed constraint formalizations.

4.3.2.4 Data dictionary

Figure 4.11 shows the stereotypes to represent data dictionaries. The `DataEntry` stereotype establishes an entry in the data dictionary. The stereotype must be applied onto a UML `DataType`. The stereotype will add to the data type the essential metadata required for DO-178C certification. Figure 4.12 illustrates the use of the stereotype to define the `DGPValue` and `HCPValue` data types used in the formalizations of HLR-2 (see Figure 4.10). `DGPValue` is an enumeration data type for the desired gear position given by the pilots. `HCPValue` is a primitive real data type for the hydraulic circuit pressure.

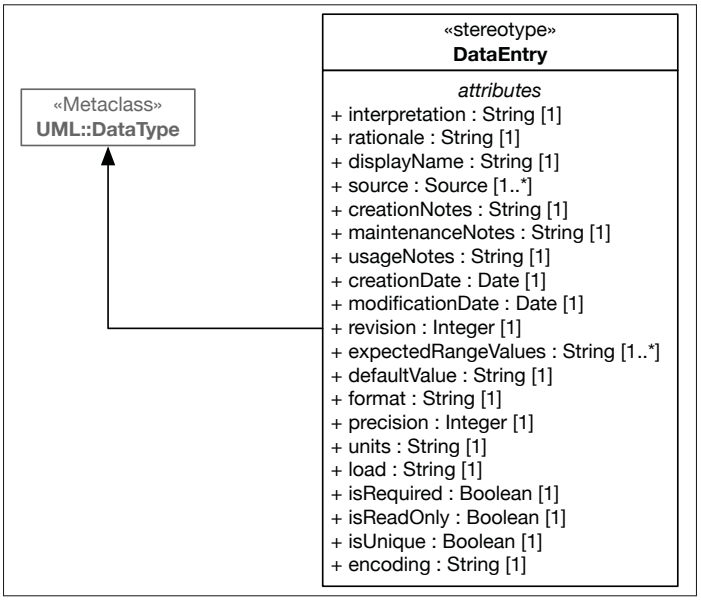


Figure 4.11 Data dictionary stereotypes.

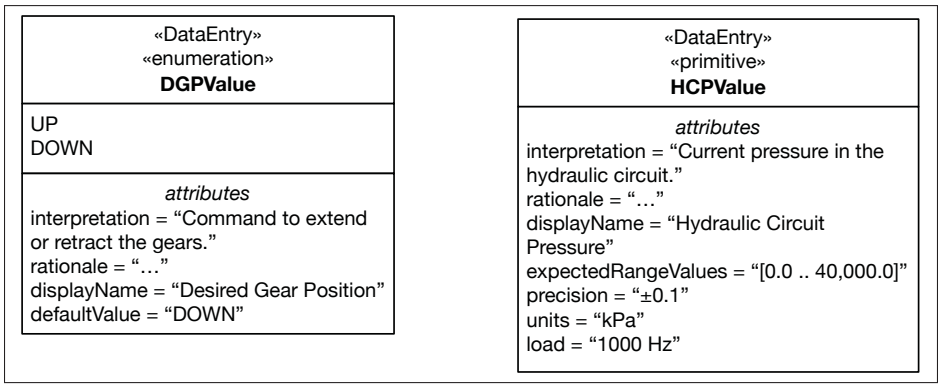


Figure 4.12 Excerpt of the LGCS specification model using SpecML showing two data entries from the data dictionary.

4.4 Reference Implementation

4.4.1 Tool support

SpecML is tool-independent, any UML modelling tool supporting UML profiles could be used to implement it. However, the UML modelling tool should: 1) support UML profile creation,

2) allow the creation of new diagram types (this is required to support the requirement specification and formalization introduced by SpecML), 3) support the validation of profile constraints to build valid models, and 4) allow the customization of messages that result from constraint violations. We developed a reference implementation for SpecML with the Eclipse Papyrus modelling environment (The Eclipse Foundation, 2017b). Papyrus was chosen because it is an open source tool and it satisfies all of the previous requirements. Furthermore, Papyrus is built on the Eclipse Modeling Framework (EMF) (The Eclipse Foundation, 2017a), which allows SpecML's integration to the *checsdm* framework. The EMF validation framework provided the means for specifying the profile's constraints and customizing the error messages given to users when a constraint is violated.

4.4.2 Overview

The reference implementation comprises three components: 1) the *profile*, 2) the *validation rules*, and 3) the *modelling tooling*. The *profile* component defines the stereotypes for the language. The *validation rules* component defines the OCL constraints for the language and utilizes Papyrus' model validation framework for their execution by the user while creating a model. The *modelling tooling* component provides the user with facilities to create a specification model, *i.e.* editor with palette and context menus, and properties view.

Figure 4.13 displays a screenshot of the SpecML reference implementation. The middle of the screen shows the model editor with a model being created. On the right of the screen is the palette with the available language constructs. On the left center of the screen is the model explorer presenting all the elements currently in the model. At the moment, the model contains one SRATS ("S" icon) and one HLR ("H" icon). The bottom of the screen displays a model validation error message indicating a violation of an OCL constraint by one of the model elements. The element causing the violation is marked in both the model editor and model explorer. The error message suggests options to the user for fixing the violation. Developer's and user's guides for SpecML's reference implementation are available in Appendix VIII and online (Paz & El Boussaidi, 2019e).

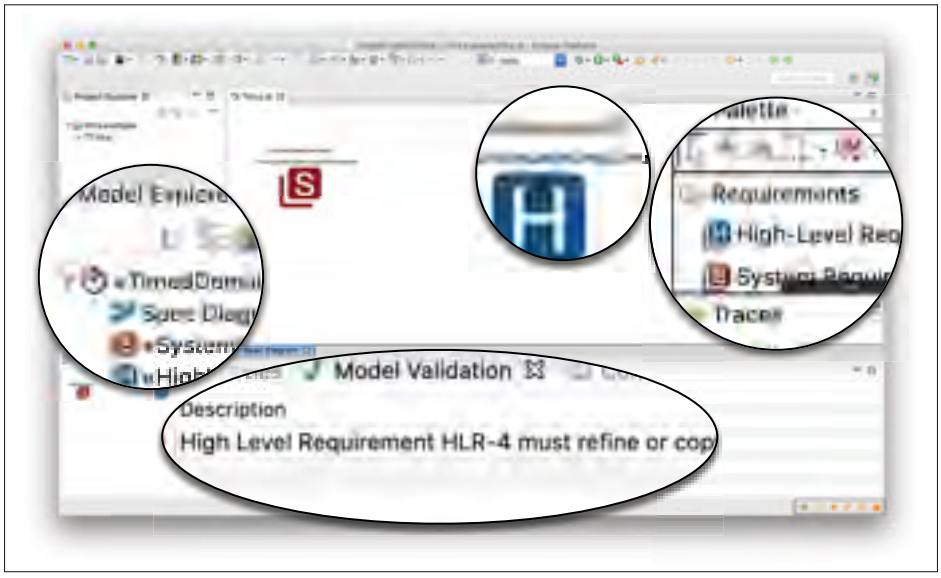


Figure 4.13 Screenshot of the SpecML reference implementation. Extracted from Paz & El Boussaidi (2019b).

Figure 4.14 presents the fragment of the specification model for the LGCS presented throughout this chapter in the reference implementation. The information for every element in the model can be accessed through Eclipse's *Properties* view.

Figure 4.15 shows an alternative tabular view of the specification model fragment for HLR-2 offered by our reference implementation. However, this is a read-only view automatically updated whenever a change is done in the specification model. Two-way update between the graphical/tree and tabular editors is left for future work. The table presented in the figure contains the reference to HLR-2 (*i.e.* SequenceControl), the IDs and texts of its formalizations, and a textual reconstruction following the format of Expression 4.1 for the conditions and predicates for each formalizing property-based statement. SpecML's reference implementation offers two additional tabular views, one of which displays only the modelled SRATS and the other displays the modelled HLRs without their formalizations but with the traceability links to their parent SRATS.

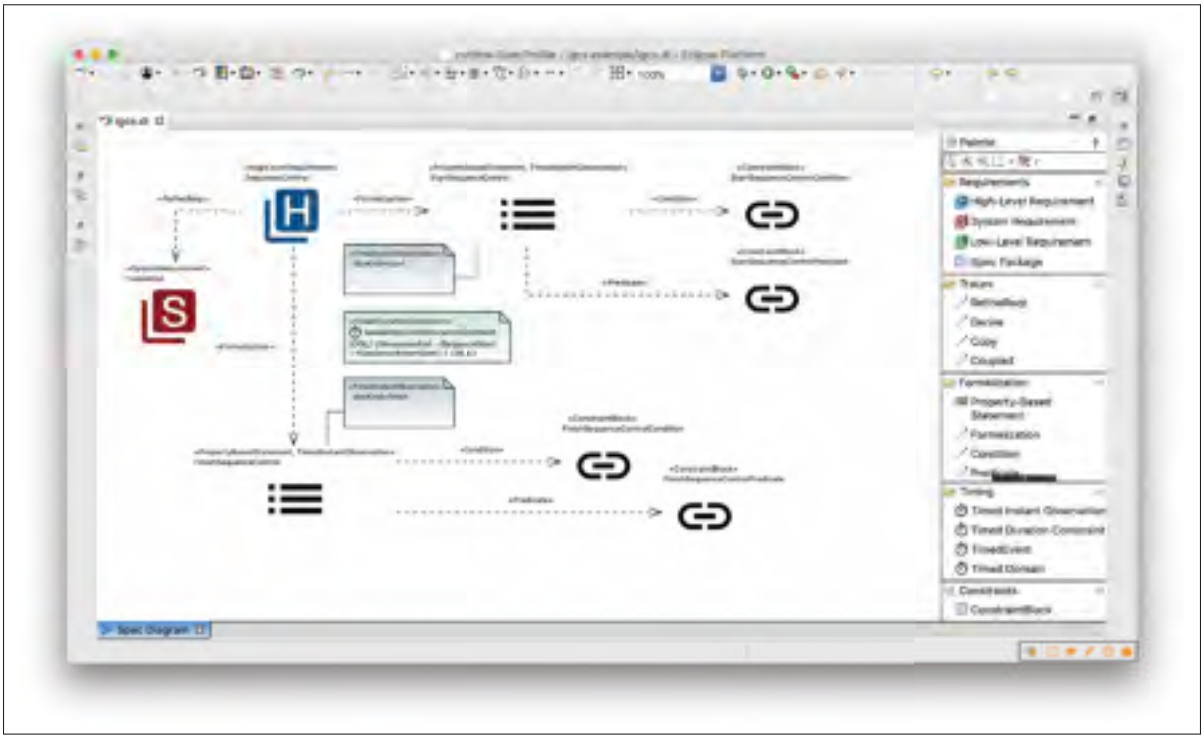


Figure 4.14 Fragment of the specification model for the LGCS in the reference implementation.

4.5 Chapter Summary

This chapter presented the unfolding and automated support of the first step of *checsdm*'s *operation* phase through SpecML, a modelling language that provides a requirements specification infrastructure for DO-178C-compliant avionics software development. SpecML offers a blended modelling approach by integrating constructs derived from an analysis of DO-178C, PBR theory, and the expression of timing constraints and clocks. Three features of SpecML were highlighted: 1) it enforces required information (*e.g.*, trace data, decomposition of requirements) for achieving objectives and activities defined in DO-178C, 2) it captures requirements in natural language to smooth the way for its adoption in industry, and 3) it provides facilities to capture requirements in a structured and semantically-rich formalism to enable requirements-based analyses and testing. SpecML was developed following an adaptation of the methodology by Selic (2007) and the extensions made by Panesar-Walawege *et al.* (2013).

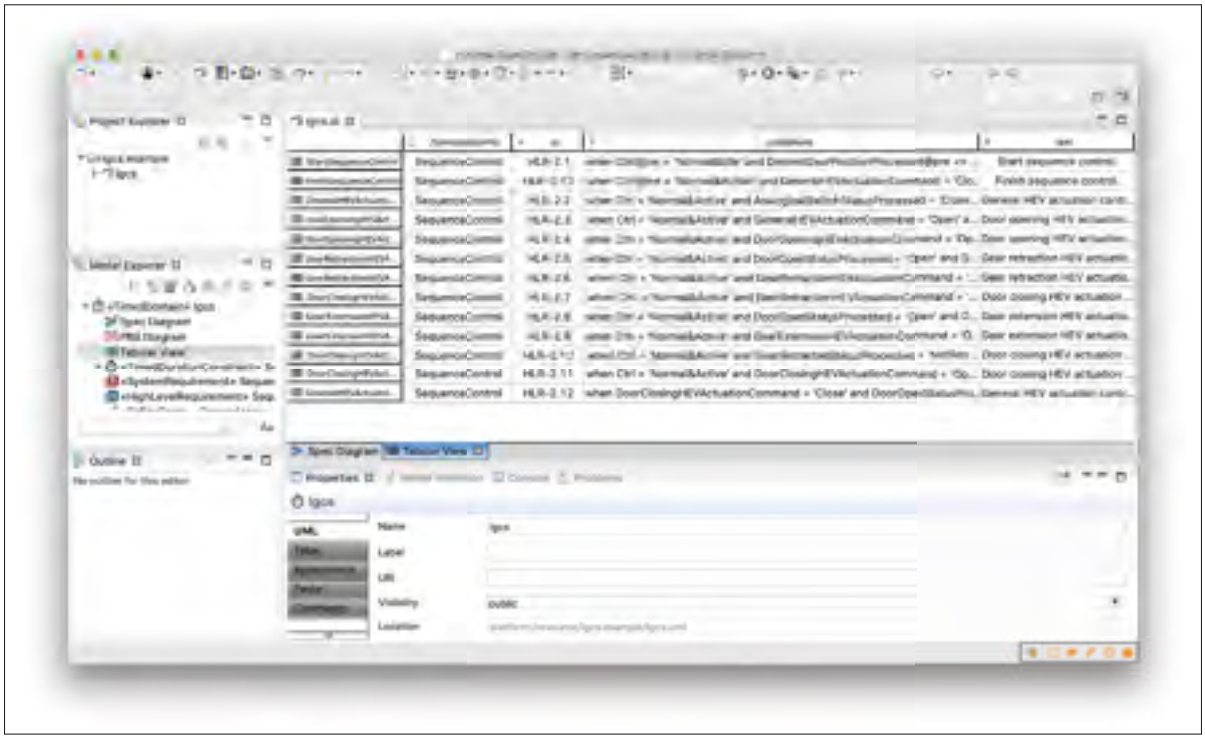


Figure 4.15 Fragment of the specification model for the LGCS in the reference implementation's tabular view.

The chapter detailed the domain metamodel behind the proposed language. The chapter then described language's architecture and the mapping between the domain metamodel's concepts and the UML metamodel. The chapter closed with a presentation of SpecML's reference implementation developed using the Eclipse Papyrus modelling environment. All elements of SpecML were illustrated with the LGCS running example.

CHAPTER 5

EVALUATION

This chapter reports on the evaluation of our proposed approach for supporting the development and certification of safety-critical avionics systems.

5.1 Research Questions

The evaluation is targeted at answering the following research questions (RQs).

RQ-1 Is it feasible to execute *checsdm*?

This research question is concerned with 1) if *checsdm* can be instantiated for a given design scenario, and 2) if the required level of effort for its execution to build an instance is acceptable. In particular for the former, the focus is on the challenges and issues encountered. For the latter, if the required level of effort is considered too high, practitioners will not adopt *checsdm*. Hence, achieving a reasonable level of effort is an important factor for a successful adoption. To answer this question we instantiated *checsdm* for two design scenarios from the industry partners. One instantiation, *checsdm4uss*, targets a design scenario involving UML, Simulink and Stateflow. *checsdm4uss* was already discussed as an example in Chapter 3. The other instantiation targets a design scenario involving AADL, Simulink and Stateflow. This instantiation is called *checsdm4a/ss* (*checsdm* for AADL, Simulink and Stateflow). These two executions of *checsdm* to derive *checsdm4uss* and *checsdm4a/ss* are taken as case studies in Section 5.2. We followed the methodology and guidelines for conducting case studies defined by Runeson & Höst (2009). A summary of *checsdm4a/ss* is presented first in the section to allow a comparison with *checsdm4uss*.

RQ-2 How does *checsdm4uss* compare to a conventional manual verification?

This research question is aimed at determining if *checsdm4uss*, derived from the execution of *checsdm*, is able to improve the identification of inconsistencies over a fully

manual verification. In particular, the focus is on the recall of inconsistencies. The answer to this question is drawn from the *checsdm4uss* case study, specifically, the execution of *checsdm4uss*' *operation* phase over three avionics systems: the LGCS introduced in Chapter 2, a Flight Control System (FCS) and an Elevator Control System (ECS) (see Subsection 5.3). The FCS and ECS were presented by Potter (2016) and Mosterman & Ghidella (2018), respectively.

RQ-3 Is it feasible to use SpecML for capturing natural language requirements, their interrelationships and formalizations that can lead to certification-compliant requirement specifications as well as for requirement-based analyses and testing?

This research question is concerned with 1) if SpecML can capture a structured, semantically rich statement of all the information found in the natural requirement statements, and 2) if required information (*e.g.*, trace data, decomposition of requirements) for achieving objectives and activities defined in DO-178C can be enforced when building a requirement specification with the language. To answer this question, we take the requirements specifications for the LGCS and the FCS (see Section 5.4). The original requirements specifications for these two systems were written in natural language. We present the results of capturing such natural language requirements, their interrelationships and formalizations as specifications models to enable further requirements-based analyses and testing.

RQ-4 Is it likely the proposed approach could be adopted in an industrial context?

Investigating the likelihood of adoption in an industrial context is a key point to evaluate the value of the proposed approach and its instantiations. To answer this question, we conducted a workshop with the industry partners to assess how domain experts perceive the proposed approach (see Subsection 5.5).

The following sections describe the setup and present the results and analysis for each research question.

5.2 RQ-1: Feasibility of *checsdm*'s Execution

5.2.1 Data collection procedure

The data collection procedure for this research question focuses on the instantiation of *checsdm*, in particular, the execution of the *elicitation* and *codification* phases, for two design scenarios of the industry partners: 1) UML, Simulink and Stateflow, which we refer to as *checsdm4uss* and described in Chapter 3.2, and 2) AADL, Simulink and Stateflow, which we refer to as *checsdm4a/ss* (*checsdm* for AADL, Simulink and Stateflow) and summarize in the following subsection. The procedure for conducting the *checsdm4uss* and *checsdm4a/ss* case studies follows closely the *checsdm* approach (see Chapter 3). Recall that in the *elicitation* phase, the requirements for the design scenarios of the industry partners were elicited. While in the *codification* phase, the support toolchains were derived from the tool framework. The answer to this research question is drawn from the efforts put in throughout the executions of these two phases to derive *checsdm4uss* and *checsdm4a/ss*. These executions were carried out iteratively. The outcomes were submitted for review to four domain experts from the industry partners, and subsequently revised.

In addition, a professional engineer working in safety-critical software development, who had two years of experience with UML, Simulink, Stateflow and EMF, was recruited to provide validation by independently executing the same two phases for the design scenario involving UML, Simulink and Stateflow. A seminar was held with the engineer to go over the *elicitation* and *codification* phases of *checsdm*, introduce the design scenario and explain the tasks to be performed. In addition, the engineer had an additional presentation to introduce and demonstrate *checsdm*'s tool framework. During the seminar, it was pointed out that keeping track of the effort put in was imperative. The engineer was then given an open time frame to carry out the task and come back with a first iteration of generated outcomes. In the course of the *elicitation phase*, the engineer had to record the mapping rules and design guidelines with given templates like the ones we used to present *checsdm4uss* in Chapter 3.

5.2.2 *checsdm4a/ss*—Another concrete instantiation of *checsdm*

We refer to the second instantiation of *checsdm* as *checsdm4a/ss* (*checsdm* for AADL, Simulink and Stateflow). In the same way as *checsdm4uss*, *checsdm4a/ss* is motivated in direct response to the industry partners' needs during their avionics system developments and certification with DO-178C, DO-331 and DO-332. This second design scenario is characterized by employing different combinations of AADL, Simulink and Stateflow design models to describe the different aspects of systems they develop. In particular, for AADL, Simulink and Stateflow, the elicited mix was motivated by the industry partners' interest to describe static system architecture in terms of distinct, communicating components with AADL, while using Simulink and Stateflow to specify each of those components' *source text*, *i.e.* their detailed design. This resulted in the following feature configuration (recall the feature model characterizing the mix of modelling languages in Figure 3.2): 1) partial coverage of the system elements, 2) provision of a structural perspective in AADL, and structural and behavioural perspectives in Simulink and Stateflow, 3) description of elements at the same and different levels of abstraction, and 4) partial overlap of elements. Regarding AADL, the industry partners use the minimal set of constructs related to processes, threads, thread groups, subprograms, devices and systems.

AADL and Simulink have a close semantic domain. Fifteen mapping rules were defined between AADL, Simulink and Stateflow. Table 5.1 lists all the mapping rules defined between these modelling languages. Similarly to *checsdm4uss*, mapping rules were established following a top-down strategy, starting from the high-level constructs (*e.g.*, AADL process and Simulink subsystem block) to lower-level ones (*e.g.*, AADL data port and Simulink block input) in order to properly capture their relationships.

Table 5.2 shows in detail mapping rule mr_as_02 describing the relationship between AADL process implementations and Simulink subsystem blocks. An AADL process implementation and a Simulink subsystem block are related *when* both of these elements have similar names. Name similarity is taken as defined in *checsdm4uss*. However, for this relationship to hold in its entirety, two *where* clauses must be satisfied. The first *where* clause, relates the AADL

Table 5.1 Summary of the mapping rules for *checsdm4a/ss*.

ID	**Name**
mr_as_01	AADL process type and Simulink subsystem
mr_as_02	AADL process implementation and Simulink subsystem
mr_as_03	AADL data port and Simulink port
mr_as_04	AADL connection and Simulink line
mr_as_05	AADL thread implementation and Simulink subsystem
mr_as_06	AADL thread group implementation and Simulink subsystem
mr_as_07	AADL subprogram implementation and Simulink subsystem
mr_as_08	AADL device implementation and Simulink subsystem
mr_as_09	AADL system implementation and Simulink subsystem
mr_as_10	AADL process type and Stateflow chart
mr_as_11	AADL process implementation and Stateflow chart
mr_as_12	AADL data port and Stateflow data
mr_as_13	AADL thread implementation and Stateflow parallel state
mr_as_14	AADL thread group implementation and Stateflow chart
mr_as_15	AADL subprogram implementation and Stateflow chart

process type for the process implementation and the Simulink subsystem block. In this clause, there is a reference to mapping rule mr_as_01 describing such a fine-grained relationship. The second *where* clause, relates the subcomponents of the AADL process implementation and the nested subsystem blocks of the Simulink subsystem. For this clause, mapping rule mr_-as_01 is also referenced but, additionally, there is a reference to mapping rule mr_as_02 as the AADL subcomponents' implementations should also be related to the nested Simulink subsystem blocks.

Table 5.2 Mapping rule mr_as_02 for AADL process implementation and Simulink subsystem.

Mapping Rule (ID: Name)	mr_as_02: AADL process implementation and Simulink subsystem
When	The AADL process implementation and the Simulink subsystem have similar names.
Where	
(1)	Process type of the AADL process implementation and Simulink subsystem block are matched (referenced rule: mr_as_01).
(2)	Subcomponents of the AADL process implementation and nested subsystem blocks of the Simulink subsystem block are matched (referenced rules: mr_as_01 and mr_as_02).

Ten design guidelines (intra- and inter-model) were developed in *checsdm4a/ss*. Five are set to be mandatory and the remaining five are recommended. Six are intra-model and four are

inter-model design guidelines. Table 5.3 lists all the design guidelines. The set of design guidelines for AADL, Simulink and Stateflow are also divided into the same four categories as those used in *checsdm4uss*. Two design guidelines, av_as_04 and av_as_06, were reused from *checsdm4uss*' design guidelines av_us_05 and av_us_18, since they concerned Simulink and Stateflow models.

Table 5.3 Summary of design guidelines for *checsdm4a/ss*.

ID	Type	Title	Priority	Category
av_as_01	Inter-	Mixed use of AADL, Simulink and Stateflow	Recommended	1
av_as_02	Intra-	Definition of a naming convention	Mandatory	2
av_as_03	Intra-	Naming of elements in AADL models	Mandatory	2
av_as_04	Intra-	Naming of elements in Simulink / Stateflow models (same as av_us_05)	Mandatory	2
av_as_05	Intra-	Naming of Simulink subsystem blocks and Stateflow charts	Recommended	2
av_as_06	Intra-	Data type of Simulink inports and outports (same as av_us_18)	Mandatory	3
av_as_07	Intra-	Parallel decomposition type for Stateflow chart and composite state	Recommended	3
av_as_08	Inter-	Expression of thread groups appearing in both AADL models and Stateflow charts	Mandatory	4
av_as_09	Inter-	Specification of AADL process as a Simulink subsystem block or Stateflow chart	Recommended	4
av_as_10	Inter-	Expression of triggers appearing in both UML and Stateflow transitions	Recommended	4

Table 5.4 shows design guideline av_as_09 providing guidance to facilitate a mapping to be established between an AADL process type and Simulink blocks/Stateflow charts. The guideline makes sure that if an AADL process type is "implemented" through a set of Simulink blocks/Stateflow charts, then these blocks/charts should be logically organized within a Simulink subsystem block having the same inputs and outputs as those defined by the AADL process type.

Regarding the codification phase, AADL modelling capabilities are supported by the Eclipse- and EMF-based Open Source AADL Tool Environment (OSATE) (Carnegie Mellon University, 2019). However, AADL model processing also required a connector tool and, thus, one was developed. This was necessary since OSATE is geared mostly towards textual development of AADL models. When the textual models are parsed, an in-memory EMF model is

Table 5.4 Design guideline av_as_09: Specification of AADL process as a Simulink subsystem block.

Guideline (ID: Title)	av_as_09: Specification of AADL process as a Simulink subsystem block
Priority	Recommended
Scope	AADL, Simulink and Stateflow
Prerequisites	None
Description	
	If an AADL process type is "implemented" through a set of Simulink blocks/Stateflow charts, then these blocks/charts should be logically organized within a Simulink subsystem block having the same inputs and outputs as those defined by the AADL process type.

Note: Some compartments are omitted to keep the table uncluttered.

generated. The developed connector tool exports the in-memory EMF model as an XMI file. Codification of both mapping rules and design guidelines is analogous to those done for *checsdm4uss*. Figure 5.1 depicts an overview of the derived toolchain.

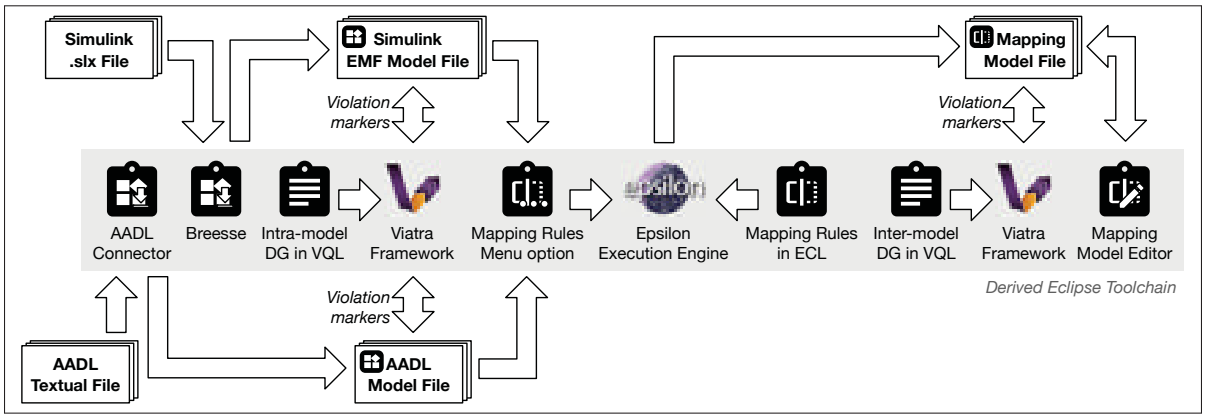


Figure 5.1 Overview of the derived toolchain for *checsdm4a/ss*.

5.2.3 Results and analysis

Chapter 3 and Section 5.2.2 describe the technical aspects of *checsdm4uss* and *checsdm4a/ss*, respectively. Thus, this subsection concentrates on providing an overview of other outcomes of the case studies without repeating any technical aspect already discussed. Table 5.5 provides a summary of *checsdm4uss* and *checsdm4a/ss* in terms of the number of mapping rules and design guidelines developed. The table also presents a summary of the outcomes generated

by the recruited engineer for the same design scenario as *checsdm4uss*. We have labeled this instantiation *checsdm4uss_alt*.

Table 5.5 Summary of the *checsdm* instantiations.
Extracted from Paz *et al.* (2020).

	checsdm4uss	***checsdm4a/ss***	***checsdm4uss_alt***
Number of mapping rules	20	15	12
Number of intra-model design guidelines	16	6	6
Number of inter-model design guidelines	5	4	1

Table 5.6 breaks down the effort involved in performing the *elicitation* phase for *checsdm4uss* and *checsdm4a/ss*. The table also includes the effort from the recruited engineer (*checsdm4uss_alt*). Regarding this phase, the challenging part was to identify the syntactical and semantical relationships between the modelling languages. Overall, the effort is reduced with the knowledge and experience of the modelling languages involved. In the case of *checsdm4a/ss*, the effort in the definition of intra-model design guidelines was reduced by reusing two design guidelines (av_us_05 and av_us_18) from *checsdm4uss*. Effort may increase with a design scenario involving a larger scope; *i.e.* a scenario demanding a larger number of modelling languages constructs to map and/or design guidelines. Iterating over this phase also increases effort. Contrarily, effort may be lower with a smaller scoped design scenario; which was the case for *checsdm4a/ss* when compared to *checsdm4uss*.

The level of effort put in by the recruited engineer for the *checsdm4uss_alt* execution was inferior to ours (see last row of Table 5.6). This was due, in part, to the fact that the engineer did not had to execute all steps from the *elicitation* and *codification* phases since the mix of modelling languages was already determined and only one iteration was expected. The set of mapping rules and design guidelines identified by the engineer were a subset of those that we identified. The engineer missed some mapping rules between certain constructs from UML (components, composite states, regions, fork and join pseudostates) and Stateflow (charts, composite states and parallel states). Other identified mapping rules referenced multiple constructs at once. This was the case for the relationships that exist between UML input/output parameters and Simulink input/output blocks. The bulk of design guidelines defined by the engineer were

intra-model. The engineer did, however, identified an inter-model design guideline equivalent to the inter-model design guideline av_us_10.

Table 5.6 Effort involved in the elicitation phases of the *checsdm* instantiations. Extracted from Paz *et al.* (2020).

	checsdm4uss	***checsdm4a/ss***	***checsdm4uss_alt***
Determination of the mix of modelling languages	1 pw	0.5 pw	n/a
Identification of mapping rules	1 pw	0.5 pw	0.5 pw
Definition of intra-model design guidelines	1 pw	0.5 pw	0.1 pw
Definition of inter-model design guidelines	1 pw	0.5 pw	0.4 pw
Refinement of elicitation outputs through iteration	2 pw	1 pw	n/a
Total effort	6 pw	3 pw	1 pw

pw: person week(s)

Table 5.7 breaks down the efforts involved in performing the *codification* phase. The table also includes the effort from the recruited engineer. During this phase, it was straightforward to translate the mapping rules into ECL match rules and the design guidelines into VQL graph patterns. Nevertheless, while codifying the *checsdm4uss* and *checsdm4a/ss* design guidelines and experimenting with them on the different avionics systems, it was found that mandatory guidelines av_us_02 and av_as_02, and recommended guidelines av_us_01, av_us_20, av_us_-21 and av_as_01 are dependent on companies and may even vary from one project to another. These guidelines are related to company design standards as well as to choices during the design process. This means they need to be codified independently for each company. Guidelines av_us_02 and av_as_02, for instance, can be codified once the naming conventions are defined.

Guidelines av_us_01 and av_as_01, for instance, describe some usage scenarios for the mixed uses of UML, Simulink and Stateflow, and AADL based on the type of system being modelled. It is not feasible to describe in these guidelines how to model all possible software and in ways that are applicable to the design processes of all organizations. Designers depend on their knowledge and past experiences for using modelling languages to represent portions of the system in ways that better suit the nature of those given portions. Thus, these guidelines are difficult to codify. For these guidelines, the engineer is more likely required to carry out a manual verification of compliance.

Like in the elicitation phase, effort in the codification phase is reduced with the knowledge and experience of the modelling languages involved as well as of the tools making up our proposed framework. Effort is also reduced with the availability of metamodels for the languages. This can be observed for *checsdm4a/ss* since the metamodels for Simulink and Stateflow were already made available by *checsdm4uss*. A minor effort was required to develop the AADL connector. Depending on the number and complexity of the mapping rules and design guidelines, the codification phase may require more—or less—effort. The recruited professional engineer had no previous experience using ECL and VQL, the two main tools in our codification framework. Indeed, mastery of these tools was not expected either after demonstrating their use during the seminar.

Table 5.7 Effort involved in the codification phases of the *checsdm* instantiations. Extracted from Paz *et al.* (2020).

	checsdm4uss	***checsdm4a/ss***	***checsdm4uss_alt***
Simulink/Stateflow metamodelling and importers	2.5 pw	0 pw	n/a
AADL connector	n/a	0.5 pw	n/a
Codification of mapping rules	2 pw	1 pw	0.7 pw
Codification of design guidelines	1.5 pw	0.5 pw	0.3 pw
Refinement of codification through iteration	2 pw	1 pw	n/a
Total effort	8 pw	3 pw	1 pw

pw: person week(s)

The bulk of the effort for *checsdm4uss* and *checsdm4a/ss* was spent on the *elicitation* and *codification* phases. These two phases required approximately three and a half person months for *checsdm4uss*, and less than one and a half person months for *checsdm4a/ss*. Most of the effort for *checsdm4uss* was spent on analyzing the modelling languages, and defining and refining the mapping rules and design guidelines. The remaining effort was spent on reviewing and codifying the Simulink and Stateflow metamodels, importers, mapping rules and design guidelines. As it can be observed with *checsdm4a/ss*, it is anticipated that a proportional amount of effort is required for other modelling languages as well. However, this is a one-time effort that needs to be put in once per set of modelling languages. Resulting mapping rules and design guidelines can then be reused across projects. Moreover, they can be reused on other *checsdm* instantiations. It is to be noted that our interest with the report of effort was not to provide a

precise record for every activity performed. The amounts of effort we documented were estimated in retrospective and, thus, should be regarded as inflated since they can be higher than they actually were.

The professional engineer carried out the instantiation in approximately a half person month (2 pw). It is to be noted that the professional engineer only carried out one iteration during the execution of *checsdm4uss_alt*, while we carried out several iterations with feedback from the industry partners. This explains the difference in terms of number of identified mapping rules and design guidelines. Nevertheless, the set of mapping rules and design guidelines identified by the engineer were a subset of those that we identified. Missing mapping rules and design guidelines could be identified in subsequent iterations and through feedback of other members of an engineering team.

5.3 RQ-2: *checsdm4uss vs.* Manual Verification

5.3.1 Data collection procedure

The data collection procedure for this research question focuses on the execution of *checsdm4uss*' *operation* phase over three software-intensive avionics systems: the LGCS, a Flight Control System (FCS) and an Elevator Control System (ECS). The LGCS was described in Chapter 2. The FCS and ECS system descriptions were developed by Potter (2016) and Masterman & Ghidella (2018), respectively, and are openly-available. The three systems were considered by the industry partners to be complex and representative of their needs. We developed the design models for these systems iteratively. These models were also presented to the four domain experts from the industry partners and afterwards revised. These models were assumed to be consistent. The design models are further described in Subsection 5.3.2.

Six professional engineers were recruited to verify the design models for consistency. The participants were separated into two groups of three engineers. The first group (named the operation group) performed the verification of the outputs of *checsdm4uss*' *operation* phase

on the three avionics systems. The second group (named the control group) performed the verification of the design models but without the assistance provided by *checsdm4uss* to act as a control group. Each participant was asked to complete a brief pre-study survey to characterize their background (see Table 5.8). Figure 5.2 summarizes the participants' background. They all have experience in software design and mostly with UML.

Table 5.8 Pre-study survey. Extracted from Paz *et al.* (2020).

Q1	Which of the following describes your experience in software design? (0 - 6 months | 6 months - 1 year | 1 - 3 years | +3 years)
Q2	Which of the following describes your experience with the Eclipse IDE? (0 - 6 months | 6 months - 1 year | 1 - 3 years | +3 years)
Q3	Which of the following describes your experience with UML? (0 - 6 months | 6 months - 1 year | 1 - 3 years | +3 years)
Q4	Which of the following describes your experience with Simulink? (0 - 6 months | 6 months - 1 year | 1 - 3 years | +3 years)
Q5	Which of the following describes your experience with Stateflow? (0 - 6 months | 6 months - 1 year | 1 - 3 years | +3 years)

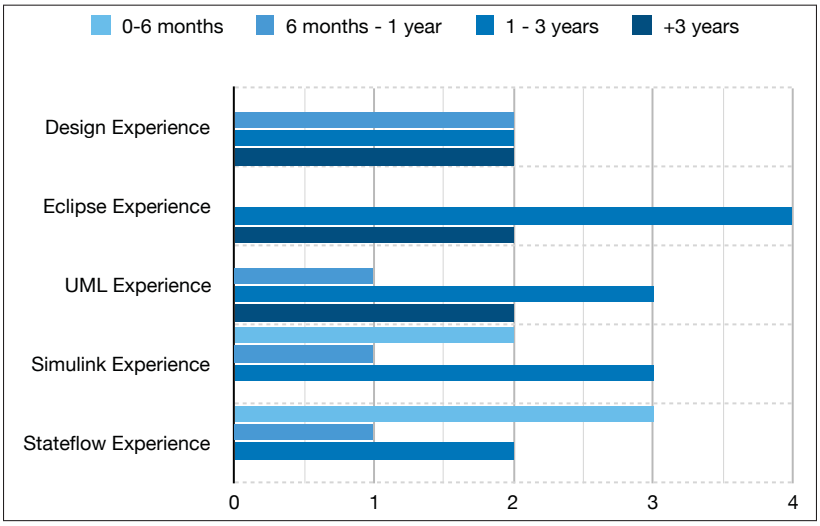


Figure 5.2 Participants' background.
Extracted from Paz *et al.* (2020).

After the pre-study survey, a seminar was held with the participants to go over the descriptions of the three avionics systems, explain the tasks they had to perform and give some preliminaries about modelling with UML, Simulink and Stateflow, as well as syntactical and semantical relationships between them. During this presentation, it was pointed out that when identifying

inconsistencies, these had to be propagated to the containing elements. The participants from the operation group had an additional fifteen-minute presentation to introduce *checsdm4uss*' *operation* phase and demonstrate how to use the support toolchain (see Figure 3.10). During this additional presentation, it was also pointed out that the toolchain may report false consistent mappings or false inconsistent mappings. Therefore, the participants were encouraged to perform a rigorous review of the results from the toolchain. Each participant of the two groups received two sets of design models (one in UML and another one in Simulink and Stateflow) from one of the avionics systems to be verified for consistency. The avionics systems were assigned randomly to the participants in each group.

The participants were then given a ninety-minute time frame to inspect the given sets of design models. For the control group, if an inconsistency was detected during their inspection, the participant had to record the inconsistency with the inconsistency report sheet in Table 5.9. For the operation group, each participant received, in addition, the design guidelines and the resulting mapping model with the flagged inconsistencies for the given set of design models. They were required to answer some questions for every mapping and inconsistency in the resulting mapping models to confirm their correctness. Table 5.10 shows a fragment example of a mapping model inspection sheet.

Table 5.9 Inconsistency report sheet. Extracted from Paz *et al.* (2020).

Inconsistency between	
UML element:	
Simulink/Stateflow element:	
Description of the inconsistency	

Finally, each participant was asked to complete a brief post-study survey to evaluate the experience. The survey for the control group included questions to rate the difficulty level of the verification task. The survey for the operation group included the same questions as the survey for the control group but it added questions to rate the difficulty level of understanding and the

Table 5.10 Fragment example of a mapping model inspection sheet. Extracted from Paz *et al.* (2020).

Mapping between	
UML input parameter:	desiredGearPosition
Simulink input:	In_DesiredGearPosition
Inconsistent:	Yes
Reasons for the inconsistency:	
Distinct types	
Q1.	If the mapping is marked as an inconsistency, is it correctly marked as an inconsistency? (Yes | No)
Q2.	If you answered Yes in Q1, is the inconsistency due to a design guideline violation? (Yes | No)
Q3.	If you answered Yes in Q2, which design guideline was violated?
Q4.	If the mapping is marked as consistent, is it correctly marked as consistent? (Yes | No)

usefulness of the assistance provided by *checsdm4uss* and its toolchain. The post-study survey is presented in Table 5.11.

Table 5.11 Post-study survey. Extracted from Paz *et al.* (2020).

Post-study survey for the control and operation groups	
Q1	How would you rate the level of difficulty to understand the tasks you were asked to perform? (Very easy | Easy | Average | Difficult | Very difficult)
Q2	How would you rate the level of difficulty to carry out the verification task? (Very easy | Easy | Average | Difficult | Very difficult)
Additional questions for the operation group	
Q3	How would you rate the level of difficulty to understand *checsdm4uss*' design guidelines? (Very easy | Easy | Average | Difficult | Very difficult)
Q4	How would you rate the level of difficulty to understand *checsdm4uss*' support toolchain? (Very easy | Easy | Average | Difficult | Very difficult)
Q5	How would you rate the level of difficulty to use *checsdm4uss*' support toolchain? (Very easy | Easy | Average | Difficult | Very difficult)
Q6	How would you rate the usefulness of *checsdm4uss* for ensuring consistency in heterogeneous design? (Very high | High | Medium | Low | None)

5.3.2 Design models

The sets of design models were created in close consultation with practitioners from the industry partners to achieve an accurate functional representation and were assumed to be consistent. All the intra- and inter-model design guidelines elicited for *checsdm4uss* were followed. Ta-

ble 5.12 provides a summary of the design models' contents in terms of the number of model element types. Before giving the design models to the participants, these were injected with design inconsistencies of different kinds (*e.g.*, naming, number of inputs/outputs, data types, source/destination of transitions) at random locations in the design models. Recall that inconsistencies in lower-level elements (*e.g.*, UML input parameters) may propagate as inconsistencies in higher-level elements (*e.g.*, UML component). The last two rows of Table 5.12 provide the number of injected inconsistencies for each system and the total number of inconsistencies taking into account propagation to container/owner elements. Figure 5.3 exemplifies an injected inconsistency between two design models of the LGCS. There is a correspondence between the input parameter `desiredGearPosition` in the UML fragment and both the output `Out_DesiredGearPosition` and input `In_DesiredGearPosition` in the Simulink fragment. However, these have different types. This inconsistency might be hard to detect. The engineer verifying the design models needs to go through several inputs and outputs on the different models and understand the syntactical and semantical relationships between the model elements.

Table 5.12 Summary of the design models for the LGCS, FCS and ECS. Extracted from Paz *et al.* (2020).

	LGCS	**FCS**	**ECS**
Number of UML components	10	10	10
Number of UML interfaces	9	8	9
Number of UML input parameters	26	30	27
Number of UML output parameters	33	32	33
Number of UML state machines	3	3	4
Number of UML states	25	6	17
Number of UML transitions	33	7	30
Number of UML choice pseudostates	0	0	2
Number of Simulink subsystem blocks	8	12	8
Number of Simulink inputs	7	31	8
Number of Simulink outputs	7	24	8
Number of Stateflow charts	3	3	6
Number of Stateflow states	23	6	19
Number of Stateflow transitions	31	6	30
Number of Stateflow junctions	0	0	2
Number of injected inconsistencies	37	46	29
Total number of inconsistencies*	84	64	45

* Taking into account propagation to container/owner elements.

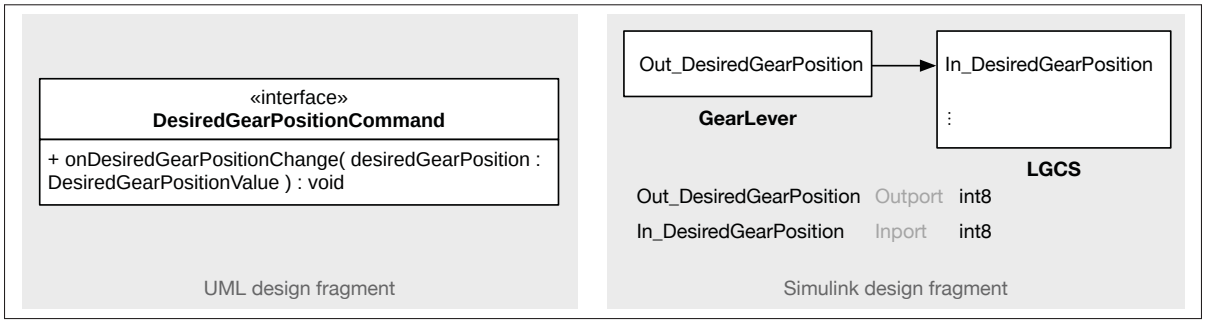


Figure 5.3 Example of an injected inconsistency used in the study. Adapted from Paz *et al.* (2020).

Table 5.13 provides a summary of the contents of the resulting mapping models for the three avionics systems in terms of the total number of mappings and inconsistency flags. Building a mapping model is a computationally demanding step that depends on the design model sizes. Computation times to generate the mapping models for the three avionics systems are presented in the last row of Table 5.13. All computations were performed on a Quad Core Intel Core i7 at 2.8 GHz with 16 GB of RAM. Computation times include applying the mapping rules, building the mapping model and cleaning up the resulting mapping model.

Table 5.13 Summary of *checsdm4uss*' resulting mapping models for the LGCS, FCS and ECS.

	LGCS	**FCS**	**ECS**
Total number of mappings	451	556	261
Number of inconsistency flags	422	546	228
Computation time	6 s	4 s	6 s

5.3.3 Results and analysis

The operation group verified the correctness of the mappings and inconsistency flags of the resulting mapping models from *checsdm4uss*. Table 5.14 provides the results of their validation by the operation group. The table reports the following:

1. The number of mappings (recalled from Table 5.13).

2. The total number of inconsistency flags (recalled from Table 5.12).

3. False consistencies, which are mappings that should have been flagged by *checsdm4uss* as inconsistent but were not.

4. True consistencies, which are mappings where the elements involved are related and consistent as confirmed by the operation group.

5. False inconsistencies, which are mappings where the elements are not related and were flagged as inconsistent.

6. True inconsistencies, which are mappings where the elements are related and were flagged as inconsistent as confirmed by the operation group.

False consistencies and false inconsistencies denote problems in the mapping rules, which may need further refinement either because some relationship in the design scenario is not being captured by some mapping rule or the current mapping rules are too restrictive.

Table 5.14 Summary of *checsdm4uss*' resulting mappings for the LGCS, FCS and ECS. Extracted from Paz *et al.* (2020).

	LGCS	**FCS**	**ECS**
Total number of mappings*	451	556	261
Number of inconsistency flags*	422	546	228
Number of false consistencies	2	0	0
Number of true consistencies	27	10	33
Number of false inconsistencies	340	482	183
Number of true inconsistencies	82	64	45
Total number of inconsistencies**	84	64	45
Precision	19%	12%	20%
Recall	98%	100%	100%

* From Table 5.13

** From Table 5.12

Precision = Number of true inconsistencies/(Number of true inconsistencies + Number of false inconsistencies)

Recall = Number of true inconsistencies/(Number of true inconsistencies + Number of false consistencies)

In the LGCS, from a total of 84 inconsistencies, the operation group confirmed the 82 mappings flagged as inconsistent in the mapping model were indeed inconsistencies. The two missing inconsistencies were falsely reported in the mapping model as consistencies. These mappings

were carried out by the application of mapping rule mr_us_03. The elements involved in the mappings had name fragments that were shared, although they were unrelated. Thus, the operation group found these to be false consistencies. This result indicates that the name similarity clause as it is defined in the mapping rules must be further refined.

Precision and recall was calculated for the tool given the validation from the operation group (see Table 5.14). There was a 98 percent recall of all inconsistencies for the LGCS, while for the FCS and ECS there was 100 percent recall. On average, the recall was 99 percent. Precision, on the other hand, was 17 percent on average. Precision for the FCS was the lowest due to the higher number of false inconsistencies reported compared to the other two systems. This result suggests that some mapping rules must be further refined. Due to the safety-critical nature of the systems, there was an aim to achieve 100 percent recall. Remember that the mapping model clean up heuristic only removes inconsistent mappings for those elements that participate in a mapping that is flagged as consistent. Any other inconsistent mapping is not discarded. This strategy had an impact on precision and caused a very high number of false inconsistencies. The participants in the operation group were able to go through all the mappings within the ninety-minute time frame. This suggests the mapping model makes it easy to go through the mappings and the number of inconsistencies was not a problem. However, the mapping rules and the mapping model clean up heuristic can be refined in order to fit specific design scenarios and ways of modelling the systems and, thus, increase precision. This will lower the number of false inconsistencies and reduce the effort involved in reviewing false mappings.

Table 5.15 provides a summary of the inconsistencies reported by the control group. All reported inconsistencies were true inconsistencies. Thus, precision for the control group was 100 percent. However, recall on the control group drops considerably, 24 percent on average. The lowest recall of the control group was for the LGCS. This system's design was slightly larger and more complex than the other two, which may explain the dip. Figure 5.4 presents a comparative view of the average recall for *checsdm4uss* mapping model and the manual work

from the control group. The figure shows that inconsistencies were identified by *checsdm4uss* with a greater recall rate over a fully manual verification.

Table 5.15 Summary of inconsistencies manually reported by the control group for the LGCS, FCS and ECS. Extracted from Paz *et al.* (2020).

	LGCS	**FCS**	**ECS**
Number of reported inconsistencies	11	21	12
Number of true reported inconsistencies	11	21	12
Total number of inconsistencies*	84	64	45
Precision	100%	100%	100%
Recall	13%	33%	27%

* From Table 5.12

Precision = Number of true reported inconsistencies/Number of reported inconsistencies

Recall = Number of true reported inconsistencies/Total number of inconsistencies

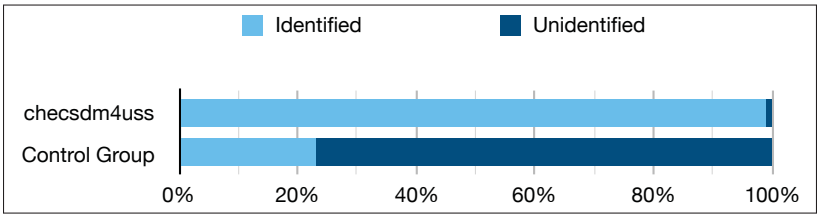


Figure 5.4 Comparison of the consolidated inconsistency recall. Extracted from Paz *et al.* (2020).

Finally, through the post-study survey, all the participants were asked to evaluate the consistency verification task they had to perform. In particular, they were asked to rate both the level of difficulty to understand the consistency verification task (Q1 of the post-study survey) and the level of difficulty to carry it out (Q2 of the post-study survey). Figure 5.5 presents the results of this survey. Most participants understood easily the verification task to be performed. However, the control group found the execution of the task to be more difficult than how the operation group found the task to be.

In a similar analysis, the participants from the operation group were asked to rate the level of difficulty to understand *checsdm4uss*' design guidelines and support toolchain, as well as its usage (Q3, Q4 and Q5 of the post-study survey). Figure 5.6 presents the results of this survey. Most of the operation group participants found *checsdm4uss*' design guidelines and support

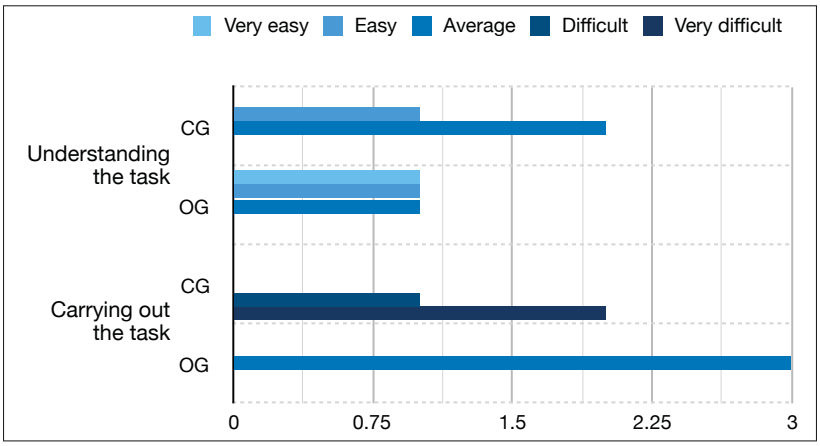


Figure 5.5 Post-study survey results for Q1 and Q2. CG: Control Group. OG: Operation Group. Extracted from Paz *et al.* (2020).

toolchain easy to understand. They, however, found the usage of the support toolchain to have an average level of difficulty. Participants reported having some difficulties navigating to the model elements involved in a mapping. Improving user experience is left for future work.

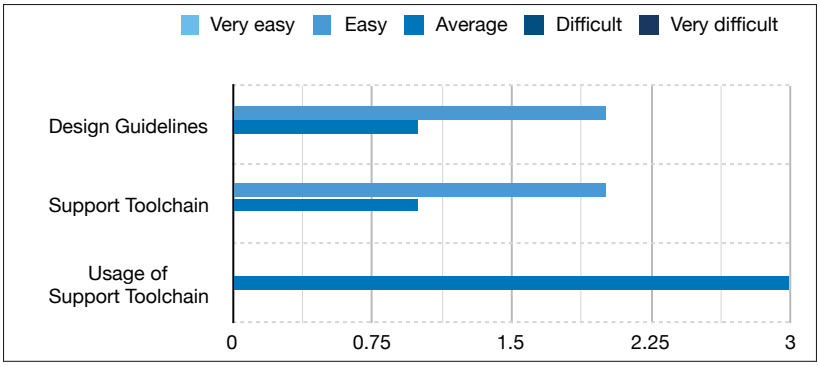


Figure 5.6 Post-study survey results for Q3, Q4 and Q5. Extracted from Paz *et al.* (2020).

To assess *checsdm4uss*' usefulness from the perspective of the participants in the operation group, they were asked to rate its support against a fully manual verification (Q6 of the post-study survey). Figure 5.7 presents this analysis. Participants found *checsdm4uss* useful and highly useful.

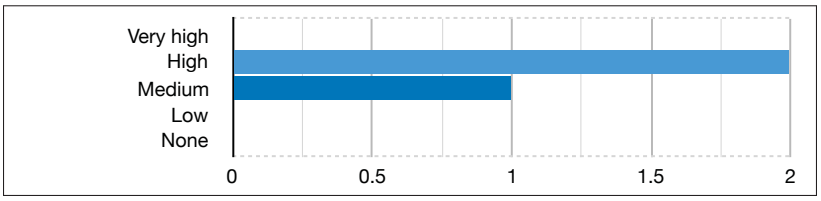


Figure 5.7 Post-study survey results for Q6. Extracted from Paz *et al.* (2020).

5.4 RQ-3: Feasibility of SpecML's Use

5.4.1 Data collection procedure

The data collection procedure for this research question focuses on the expression of the different requirements for the LGCS and the FCS using SpecML. Recall that SpecML captures a blended specification of requirements: natural language statements and formalized statements of the former. The latter were created by translating the natural language statements into PBRs and annotating such PBRs with time-dependent constraints where necessary. The entire process of using SpecML to specify the requirements of the LGCS and the FCS was iterative. The outcomes were reviewed by four domain experts from the industry partners and were subsequently revised.

The LGCS was described in Chapter 2. The FCS was developed by Potter (2016) and openly distributed online. The FCS is the system responsible for providing attitude (*i.e.* the aircraft's orientation about its center of mass) and attitude rate control based on pilot input commands to keep them within the flight envelope of the aircraft. The FCS controls three hydraulic actuators that allow the aircraft to pitch up or down, roll right or left, and yaw right or left. The ECS, which is used during the evaluation of *checsdm4uss*' operation phase, was not considered for this part of the evaluation due to lack of time in the project.

For the LGCS, there are eighteen SRATS (identified by the prefix SRATS- and a unique number), which have been refined into eighteen HLRs (identified by the prefix HLR- and a unique number). For the FCS, there are eleven SRATS (identified by the prefix SR_ and a unique

number), which have been refined into thirteen HLRs (identified by the prefix HLR_ and a unique number). The sets of requirements for these two systems cover different kinds of functional elements from one another. For instance, the LGCS includes requirements involving the sequencing of property evaluations, which the FCS does not. The FCS, instead, includes requirements involving continuous modulated control through a proportional–integral–derivative loop. For the sake of brevity only one SRATS and its refining HLR are discussed for each system. Table 5.16 presents the requirements extracted from these systems to illustrate the requirements modelling using SpecML.

Table 5.16 Requirements extracted from the LGCS and FCS.

System	Requirement Type	Requirement Statement
LGCS	SRATS	**SRATS-2 Retraction Sequence.** When the pilots switch the gear lever to `Up`, the LGCS shall retract the gears in under 28 seconds.
LGCS	HLR	**HLR-2 Retraction Sequence Control.** When the LGCS receives an `Up` value for the `Desired Gear Position`, the LGCS shall carry out the retraction sequence in under 28 seconds.
FCS	SRATS	**SR_4 Hydraulic Actuator Control Loop Performance.** The FCS shall control the hydraulic actuator position with a minimum bandwidth of 10 Hz and a minimum damping of 0.4.
FCS	HLR	**HLR_4 Hydraulic Actuator Loop Control.** Each hydraulic actuator loop shall be implemented as a PID (proportional-integral-derivative) control loop operating at a 1 ms frame rate. The proportional gain shall be 0.3. The integral gain shall be 0.12. The derivative gain shall be 0.02.

5.4.2 Results and analysis

Figure 5.8 presents a fragment of the LGCS specification model in SpecML using the SysML notation. Recall HLR-2 describes the behaviour of the LGCS when a desired gear position is given by the pilots. There are a series of actions the LGCS must perform as part of this requirement and a single formalization statement is not sufficient to cover the whole behaviour. Thus, thirteen property-based statements are introduced to formalize the HLR. The figure shows only the first and last of these formalization statements (2.1 and 2.13). Formalization 2.1 reads as follows: *when the LGCS is functioning normally and is idle, and there is a change in the desired gear position after it has been validated, then the LGCS shall become active*. Formal-

ization 2.13 reads as follows: *when the LGCS is functioning normally and is active, the general hydraulic electro-valve has been set to close, the hydraulic circuit pressure is validated to be under* $30,000\,kpa$*, and the analogical switch status is validated to be open, then the LGCS shall become idle*. These two statements are annotated as timed instant observations. A timed duration constraint is placed on these two timed instant observations to restrict the temporal distance between the two events to be less than 28 seconds.

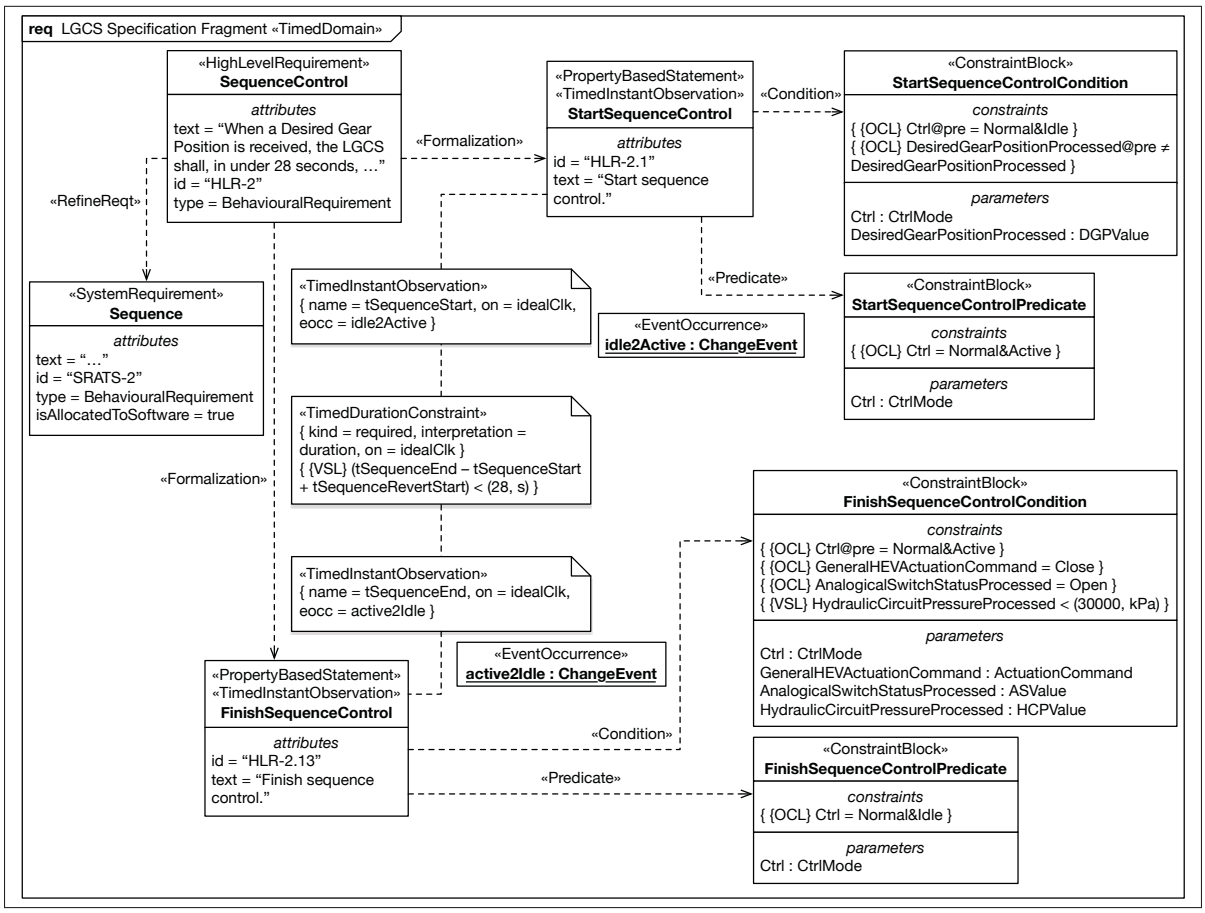


Figure 5.8 Fragment of the specification model for the LGCS.

Figure 5.9 presents the complete formalization of HLR-2 in SpecML's reference implementation. Although we were able to successfully captured all the LGCS' requirements with SpecML and using its reference implementation, the figure does put in evidence a scalability problem. As the number of requirements and their formalizations we had modelled started to grow, we

experienced that the graphical view can become unreadable and cumbersome to work with. Therefore, we developed three tabular views (*i.e.* modelled SRATS, modelled HLRs without their formalizations but with traceability links to parent SRATS, and modelled formalizations for the HLRs) to mitigate such a scalability problem. However, the tabular views are read-only. Once a tabular view is created it will be automatically updated whenever a change is done in the specification model. Two-way update between the graphical/tree and tabular editors is left for future work.

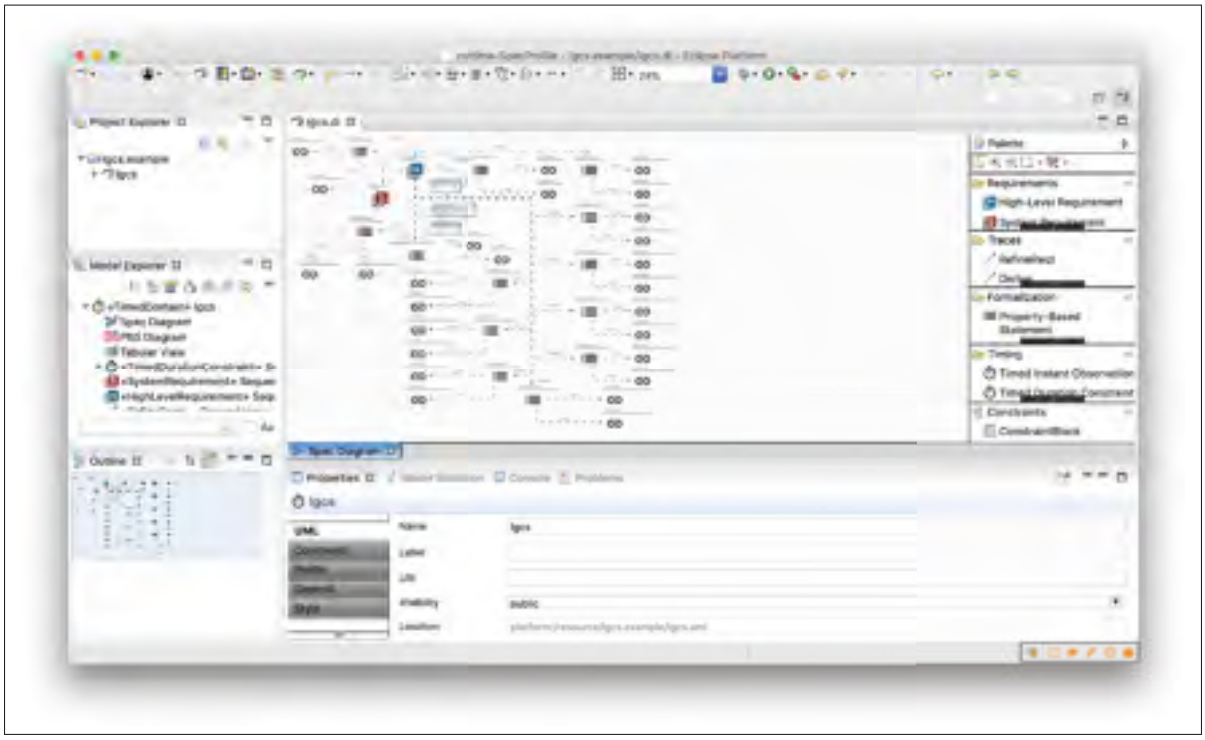


Figure 5.9 Complete formalization of the LGCS' HLR-2 in SpecML's reference implementation.

Figure 5.11 presents the specification of SR_4 and its refinement by HLR_4 from the FCS using the SysML notation. Figure 5.12 shows its equivalent using the SpecML reference implementation. The system requirement SR_4 is a system requirement allocated to software, which is refined by HLR_4. HLR_4 describes the behaviour of the hydraulic actuator loop control. There are three hydraulic actuator loops, however, in this case a single formalization statement is sufficient to cover the whole behaviour. Thus, only one property-based statement

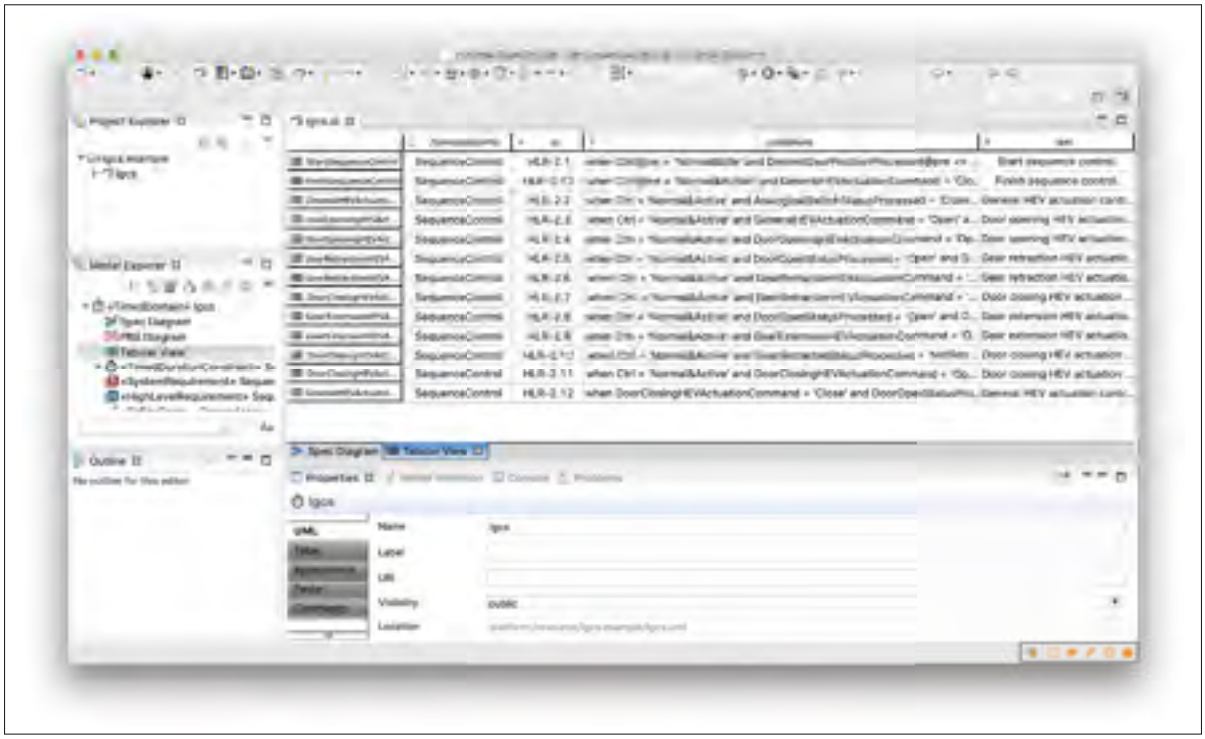


Figure 5.10 Tabular view for the complete formalization of the LGCS' HLR-2 in SpecML's reference implementation.

is introduced to formalize the HLR. The formalization statement reads as follows: *the FCS shall implement the actuator command loop as a PID control loop*. The predicate of this statement has nested SysML constraint blocks that defined separately each term of the PID control loop. Furthermore, the formalization statement is annotated as a timed event that occurs every 1 millisecond for an indefinite number of repetitions.

Producing the requirement specification model of the LGCS and FCS with SpecML is only an intermediate goal. The requirements are allocated to entities of the design models intended to satisfy them. SysML parametric diagrams can include usages of the constraint blocks formalizing the requirements to constrain the properties of the entities in the design models. For instance in the FCS, binding the parameters of the `CommandLoopControlPredicate` constraint block to the specific properties of the `PitchActuatorLoop`, `RollActuatorLoop` and `YawActuatorLoop` design entities (not shown) intended to satisfy HLR_4, will provide the

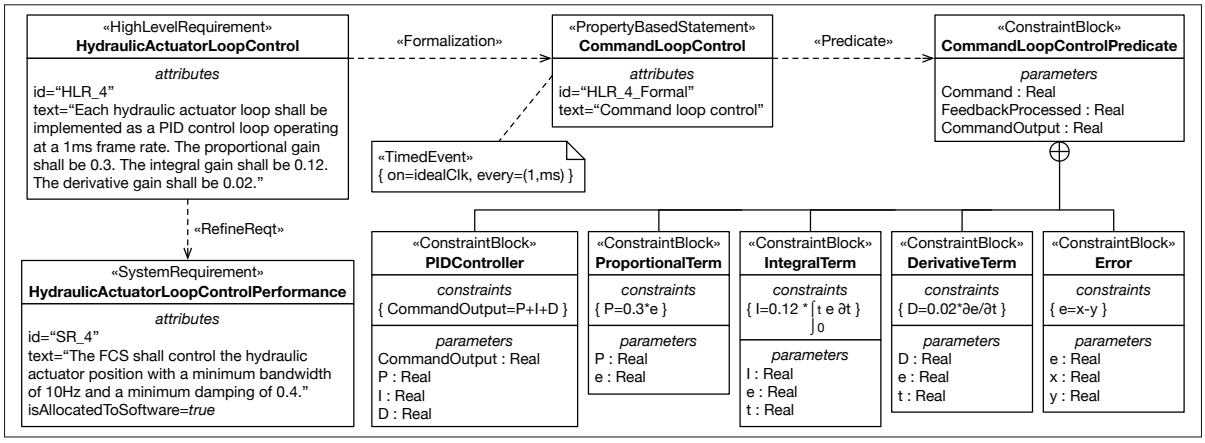


Figure 5.11 Fragment of the specification model for the FCS. Extracted from Paz & El Boussaidi (2019b).

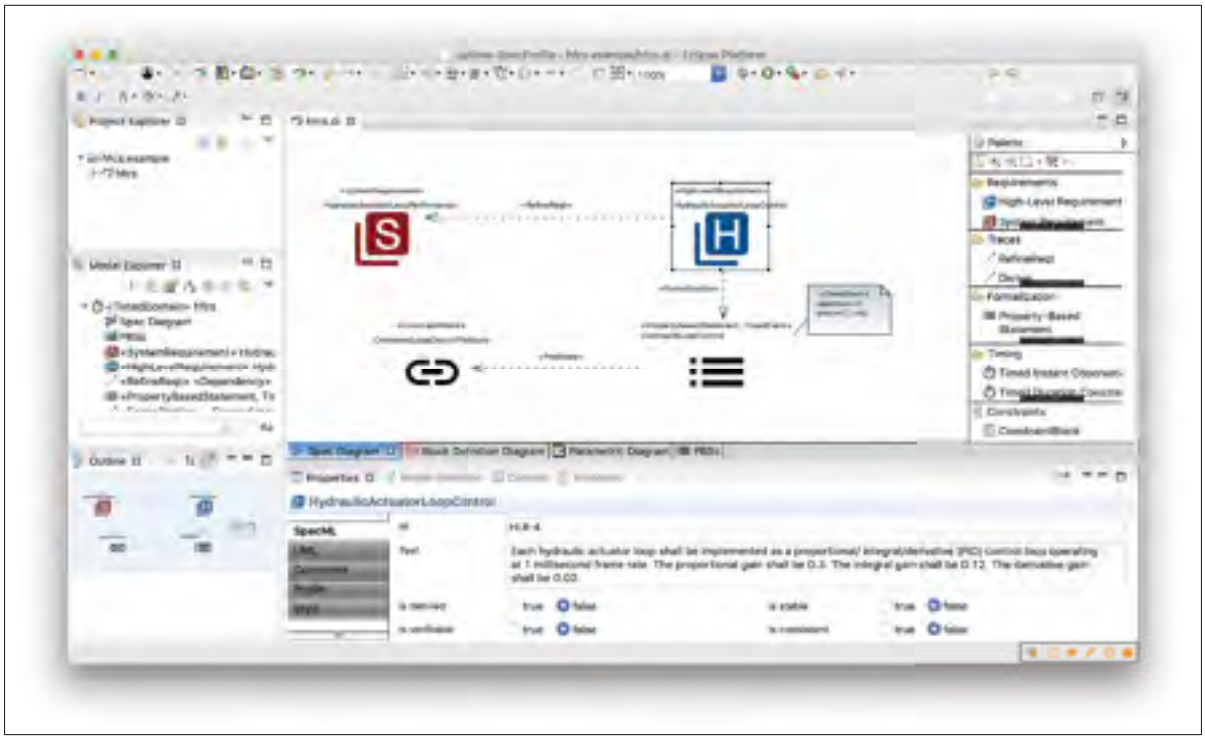


Figure 5.12 Screenshot of the specification and formalization of HLR_4 for the FCS with the SpecML reference implementation. Extracted from Paz & El Boussaidi (2019b).

values for the parameters. This modelling establishes the method of evaluating design compli-

ance with the specified requirements. Moreover, the property-based statements can be used to generate test cases. Figure 5.13 shows a SysML parametric diagram detailing the usage of the nested constraints for the PID control loop in HLR_4 of the FCS. This parametric diagram can be used to bind its parameters to the specific properties in a design model.

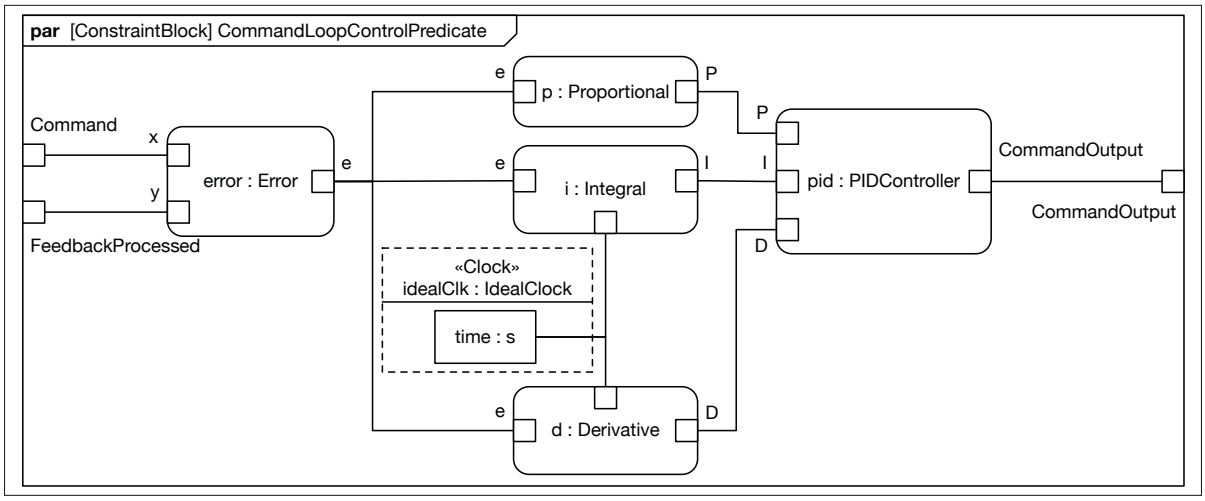


Figure 5.13 SysML parametric diagram of the `CommandLoopControlPredicate` constraint block.

The analysis of the previous results focuses on the benefits experienced in the process. Four benefits of using SpecML can be highlighted. The first benefit regards requirement specification in accordance with regulation. Using SpecML, all the LGCS and FCS requirements (SRATS and HLRs) were modelled in a hierarchical way along with their interrelationships satisfying DO-178C objectives. The second benefit is that SpecML can relieve requirements engineers from the error-prone and labor-intensive work of manually verifying every requirement for compliance with DO-178C objectives. While building the specification model, several errors related to deviations from DO-178C objectives were detected with the reference implementation and corrected accordingly. The third benefit pertains to facilitating communication between stakeholders by capturing requirements in natural language while still allowing requirement formalization to enable analyses and testing. All the HLRs for the LGCS and FCS were formalized using the specialized stereotypes for such a purpose. As with any model,

SpecML does not actually perform requirement analyses or testing, it is intended to provide facilities that enable requirements-based analyses and testing.

It is to be acknowledged requirements specification standards vary from one company to another. This results in different ways of writing the same requirement statement. The fourth benefit is that SpecML can accommodate the review efforts already put in place to check compliance with company-defined requirement standards. Requirement statements in SpecML are first represented in natural language and must be clearly defined. The quality of the requirements is dependent on such descriptions since they will be translated into PBR statements and timing constraints.

5.5 RQ-4: Likelihood of industry adoption

5.5.1 Data collection procedure

The workshop was conducted with the industry partners with the goal of investigating our proposed approach's likelihood of being adopted in a real industrial context (RQ-4). This investigation was carried out following the goal-question-metric (GQM) approach (Basili *et al.*, 1994). Six practitioners attended the workshop. However, one of them was not present for the entire workshop. The group was hand-picked by two champions from the industry partners, who had knowledge of the approach. The group included senior engineers with ample experience in the development and certification of safety-critical avionics software, who would likely produce valuable feedback on the approach.

At the beginning of the workshop, each attendee practitioner was asked to complete a brief pre-workshop survey (see Table 5.17) to characterize their background. Based on the responses obtained in the pre-workshop survey, all the practitioners had over five years experience with safety-critical system development. Requirements specifications, software design and verification were integral parts of their jobs. For all but one practitioner, certification was also an important aspect of their jobs.

Table 5.17 Pre-workshop survey.

Q1	Which of the following describes your experience in safety-critical software development? (0 - 12 months | 1 year - 3 year | 3 - 5 years | +5 years)
Q2	Is requirement specification an important aspect of your job? (Yes | No)
Q3	Is software design an important aspect of your job? (Yes | No)
Q4	Is verification an important aspect of your job? (Yes | No)
Q5	Is certification an important aspect of your job? (Yes | No)

The workshop then lasted for one hour. It proceeded with an interactive presentation giving a thorough look into the phases of the *checsdm* approach, the SpecML modelling language and providing as example the *checsdm4uss* concrete instantiation. Afterwards, a demonstration of the toolchain was given. Throughout the presentation and tool demonstrations the attendees could ask questions to clarify any concerns. Following the presentation session, a post-workshop survey with the GQM model questions was circulated to the practitioners. Responses were anonymous and were based on the practitioners' perceived advantages.

Table 5.18 shows the complete GQM model. We divided the questions in the GQM model into three groups. The first group, Q1 and Q2, was about the characterization of our proposed approach's comprehensibility. The second group, Q3 and Q4, was about the evaluation of our proposed approach's overall value. The last group, Q5–Q7, was about the evaluation of our proposed approach's characteristics that are relevant with respect to the issue (*e.g.*, support for certification).

5.5.2 Results and analysis

It was found that four attendee practitioners perceived the *checsdm* approach and its instantiation (*checsdm4uss*) as being easy to understand (see Figure 5.14). One practitioner thought they were averagely understandable. No one found them to be difficult to understand. It was found that all six attendee practitioners perceived SpecML easy to understand (see Figure 5.15). Re-

Table 5.18 GQM model for the assessment workshop.

Goal	**Purpose**	Assess
	Issue	the likelihood of adoption of
	Object	the proposed approach
	Viewpoint	from the practitioner's viewpoint
Question	**Q1.**	Were (1) *checsdm*, (2) *checsdm4uss* and (3) SpecML easy to understand?
Metric	**M1.**	Rating score (for each numeral) (Very easy | Easy | Average | Difficult | Very difficult)
Question	**Q2.**	Would you use (1) *checsdm*, (2) *checsdm4uss* and (3) SpecML to help in your work?
Metric	**M2.**	Rating score (for each numeral) (Definitely | Likely | Not sure | Not likely | Definitely not)
Question	**Q3.**	Do you see value in adopting (1) *checsdm* and (2) *checsdm4uss* for ensuring consistency of heterogeneous design models?
Metric	**M3.**	Rating score (for each numeral) (Definitely | Likely | Not sure | Not likely | Definitely not)
Question	**Q4.**	Do you see value in adopting SpecML for supporting (1) requirement specification, (2) requirement-based testing, and (3) certification efforts?
Metric	**M4.**	Rating score (for each numeral) (Definitely | Likely | Not sure | Not likely | Definitely not)
Question	**Q5.**	Does the resulting mapping model in the *operation* phase provide useful assistance for reviewing and solving consistency issues in heterogeneous design models?
Metric	**M5.**	Rating score (Definitely | Likely | Not sure | Not likely | Definitely not)
Question	**Q6.**	Do you find the resulting requirements specification model simple enough for use when communicating with a certification agent?
Metric	**M6.**	Rating score (Definitely | Likely | Not sure | Not likely | Definitely not)
Question	**Q7.**	Does the proposed approach provide useful assistance for adhering to certification compliance needs?
Metric	**M7.**	Rating score (Definitely | Likely | Not sure | Not likely | Definitely not)

garding adoption of the approach to help in their work, five practitioners agreed that *checsdm* and *checsdm4uss* were worth adopting (see Figure 5.16). Regarding adoption of SpecML to help in their work, four attendees thought there was likelihood while the remaining two were not sure (see Figure 5.17).

In terms of adopting the approach for the specific task of ensuring consistency of heterogeneous design models, three practitioners were definitely in favor of adopting the approach and one of them thought there was likelihood (see Figure 5.18). One practitioner was not sure. In terms of value perceived for supporting requirement specification, five practitioners were in favor of adopting SpecML (see Figure 5.19). One of the attendees thought there was little like-

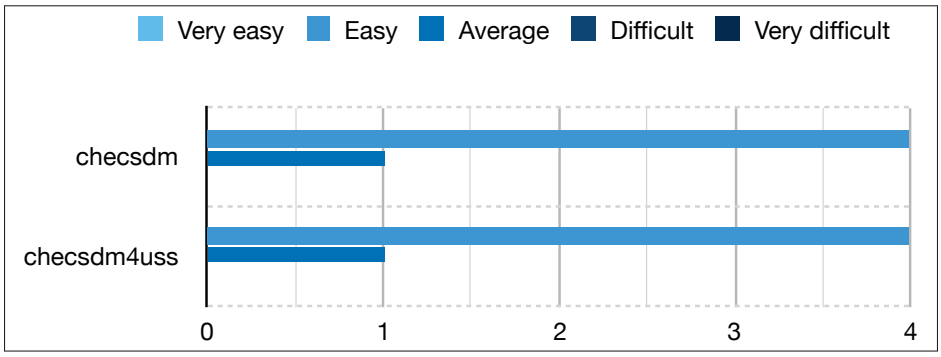


Figure 5.14 Q1. Were (1) *checsdm* [and] (2) *checsdm4uss* easy to understand?

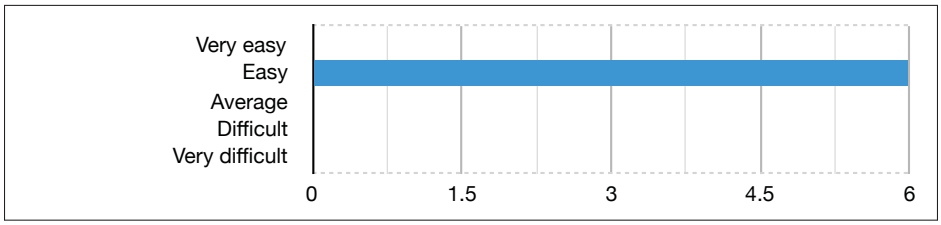


Figure 5.15 Q1. [Was] (3) SpecML easy to understand?

lihood. About value perceived for supporting requirements-based testing, all six practitioners were in favor of adopting it (see Figure 5.19). With respect to value perceived for supporting certification efforts, four participants were in favor of adopting the language (see Figure 5.19). The remaining two were not sure.

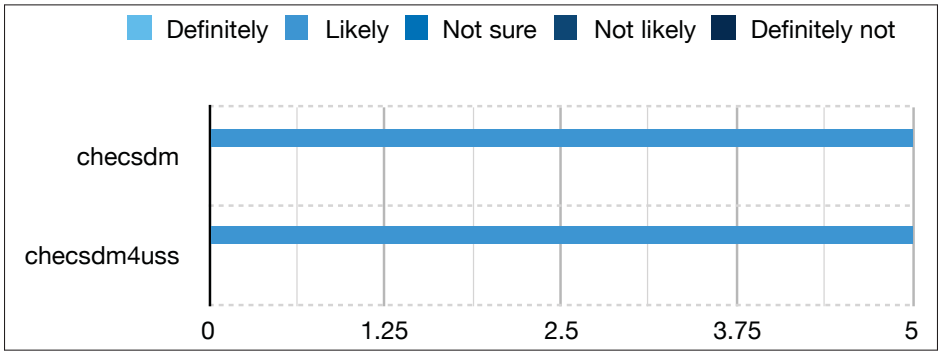


Figure 5.16 Q2. Would you use (1) *checsdm* [and] (2) *checsdm4uss* to help in your work?

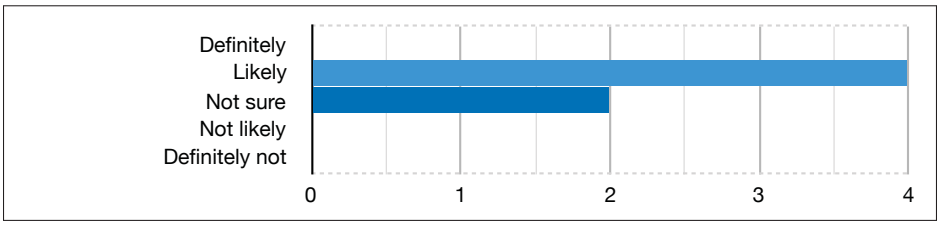


Figure 5.17 Q2. Would you use (3) SpecML to help in your work?

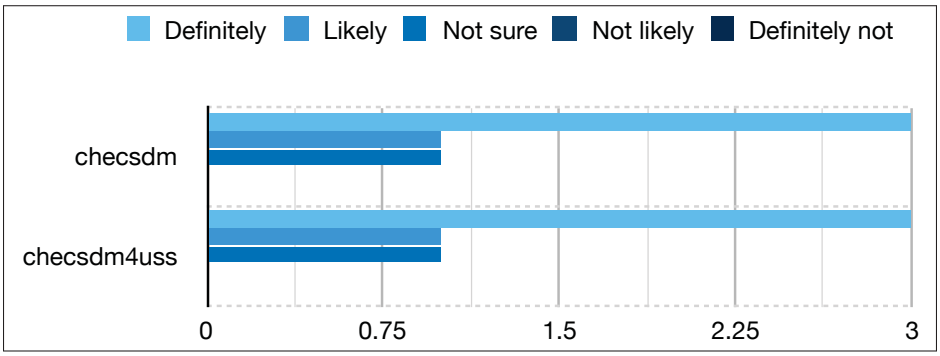


Figure 5.18 Q3. Do you see value in adopting (1) *checsdm* and (2) *checsdm4uss* for ensuring consistency of heterogeneous design models?

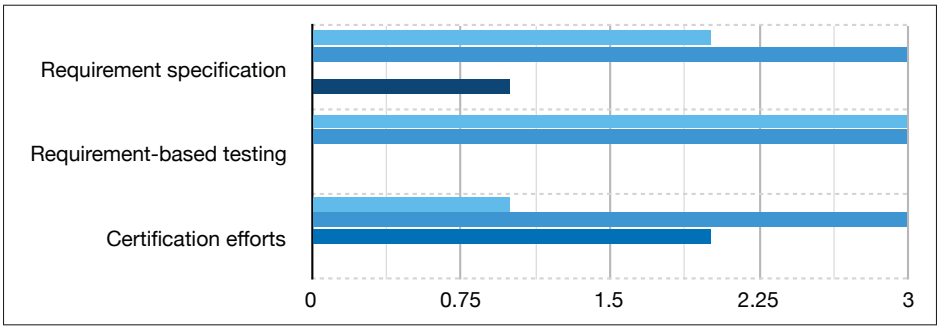


Figure 5.19 Q4. Do you see value in adopting SpecML for supporting (1) requirement specification, (2) requirement-based testing, and (3) certification efforts?

About the usefulness of the mapping model in assisting the review and resolution of consistency issues in heterogeneous design models, three practitioners definitely perceived its usefulness, while the other two saw some usefulness (see Figure 5.20). Reuse across multiple

projects following the same design scenario, short execution times of codified design guidelines and mapping rules, and higher recall rates than a manual design model verification were the driving benefits for their responses. Indeed, the one-time overhead introduced by the elicitation and codification phases was their main wavering factor.

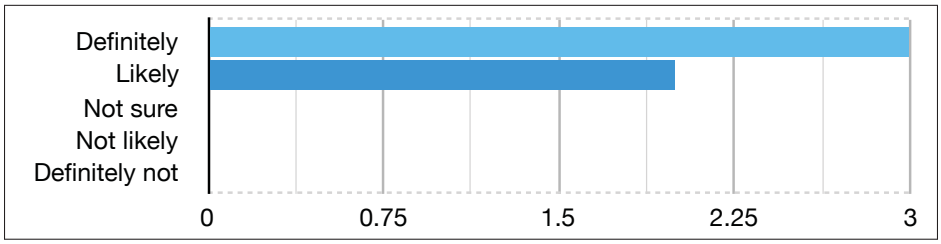


Figure 5.20 Q5. Does the resulting mapping model in the *operation* phase provide useful assistance for reviewing and solving consistency issues in heterogeneous design models?

Concerning the simplicity of the resulting specification model and using it when communicating with a certification agent, one participant was definitely in favor and two thought there was likelihood (see Figure 5.21). The remaining three participants were not sure. In the matter of perceiving useful assistance for adhering to certification compliance needs, all six practitioners saw usefulness (see Figure 5.22).

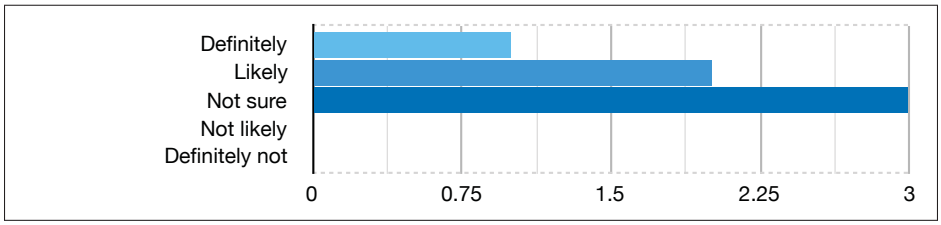


Figure 5.21 Q6. Do you find the resulting requirements specification model simple enough for use when communicating with a certification agent?

We also received some open comments from the participants about our proposed approach. They see the approach beneficial, in particular, for aiding in dissimilar coding and in an independent test generation chain branching from the design. For the latter, the practitioner

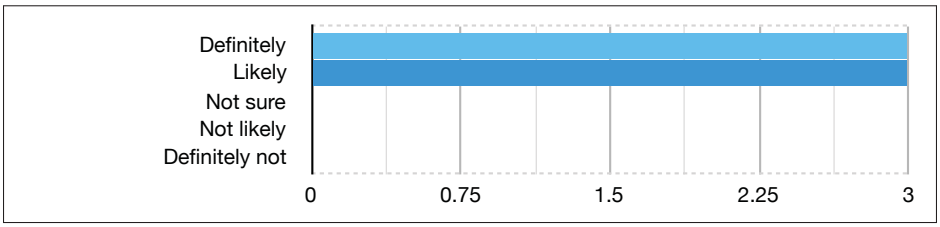


Figure 5.22 Q7. Does the proposed approach provide useful assistance for adhering to certification compliance needs?

explained the design could be done using one modelling language, then an additional independent design could be created using a different modelling language consistent with the first (as verified by a *checsdm* instantiation) and from which tests could be generated. At the end, the evidence of consistency (provided by the *checsdm* instantiation) between both designs along with the results from the generated tests' executions could be presented to the certification authority. Regarding specific feedback on SpecML, one practitioner made two feature requests: 1) allowing the import of requirements from IBM DOORS, and 2) allowing the export of requirements as a document. For the practitioner, such features would greatly complement the tooling to facilitate certification. Another practitioner would like to see SpecML's support to be adapted to include support for other norms in addition to DO-178C, like ISO 26262. Extending the proposed approach in the suggested regards is left for future work.

In summary, the responses from the practitioners suggest an overall likelihood of adopting the proposed approach in an industrial context as they clearly see advantages in using the approach within the context of their work. The practitioners took into account several factors to determine if the efforts involved were acceptable. These were 1) the effort invested in the *elicitation* and *codification* phases for a set of modelling languages, 2) perceived impact on supporting the satisfaction of DO-178C verification objectives, and 3) side benefits that the mapping rules, design guidelines and the resulting mapping model could bring, for instance, to subsequent development activities. The involved practitioners from the industry partners, overall, considered the efforts for these two phases on *checsdm4uss* and *checsdm4a/ss* to be

acceptable as they are one-time efforts that can be compensated with the number of projects making use of them over time.

5.6 Threats to Validity

Certain factors may have influenced the evaluation's results. This section briefly discusses these factors and the way they were mitigated.

5.6.1 Internal validity

Threats to internal validity have to do with factors that might affect the results of the evaluation. One of these factors is establishing *checsdm*'s feasibility argument on two case studies, *checsdm4uss* and *checsdm4a/ss*. It is acknowledged that for a different, perhaps more complex, design scenario there could be a greater number of—and possibly more complex—mapping rules and design guidelines. Nonetheless, this threat is mitigated by the fact that both *checsdm4uss* and *checsdm4a/ss* were considered by the industry partners to be representative design scenarios of their needs.

Another factor that might affect internal validity is the set of design models on which the *operation* phase of *checsdm4uss* was executed. This set was small and each design model had to be manageable in size and complexity in order to carry out the case study with the recruited professional engineers. However, the design models were validated through in-depth discussions with practitioners from the industry partners, who have ample experience in both avionics software development and DO-178C certification. These practitioners see these models as representative of their own design models. Inconsistencies injected into these design models may not simulate entirely those that a designer might introduce when creating a design model. Nevertheless, this threat was mitigated by injecting various kinds of inconsistencies at random locations of the design models.

Accuracy of the Simulink and Stateflow metamodel may have an impact on the design models. In order to lessen this threat, existing knowledge about Simulink constructs was reused from

Massif's (IncQuery Labs, 2017) Simulink metamodel. In addition, metamodelling was carried out iteratively and with validation from industry practitioners. The effectiveness of the mapping strategy is also a significant factor affecting the results. The current codification of mapping rules and mapping model clean-up heuristic result in identifying consistency issues that may not be consistency issues at all and could, perhaps, be intentional. Codification could have been implemented differently. Nevertheless, the industry partners find reviewing these mappings to be worthwhile. Such a situation reinforces the purpose of the *elicitation* phase.

Selection bias of participants in both case studies and the workshop is a noteworthy factor to internal validity. Regarding the *checsdm4uss* case study, professional engineers knowledgeable about software design were selected. Differences between their levels of experience was used to direct their assignment to either the operation or the control groups. The intention with this was to make sure that these two groups could be seen as homogeneous between them as possible, *i.e.* have the same balance of experienced and less-experienced engineers. Regarding the *elicitation* and *codification* phases of *checsdm4uss_alt*, a professional engineer in the field of avionics software development was recruited. The engineer knew and had experience with the modelling languages, however, not with all the modelling technologies behind the *checsdm* tool framework. Hence, the demonstration of the tool framework during the seminar. Regarding the workshop with practitioners, engineers knowledgeable about the challenges of heterogeneous design were selected.

Most participating engineers in the *checsdm4uss* case study had three years or more in experience with UML. Their experience with Simulink and Stateflow, however, varied more. In order to ensure the minimum knowledge about the modelling languages required for the case study, they were given a briefing seminar. The seminar went over the descriptions of the three avionics systems, explained the tasks they had to perform and gave the necessary preliminaries about modelling with UML, Simulink and Stateflow, as well as syntactical and semantical relationships between them. Particularly for the control group, which was asked to manually analyze the design models, their precision for inconsistency identification was of 100 percent and a recall rate of 24 percent, on average. Precision and recall would be expected to drop

with less experienced participants, while the recall rate would be expected to rise with more experienced participants.

A factor threatening internal validity for SpecML is establishing its feasibility argument on two avionics systems. It is acknowledged that for different requirements specifications, more complex and following different requirements standards could lead to more complex specification models. Nonetheless, the LGCS and FCS were considered by involved practitioners from industry to be complex and representative of their practices.

Another factor that might affect internal validity is that SpecML's requirement formalization approach is primarily based on the property-based requirement (PBR) theory (Micouin, 2008). This is a fairly recent theory of requirement specification and more studies are needed to widely identify its limitations about the kinds of requirements that can be effectively specified as PBRs. However, the LGCS and FCS do contain different kinds of functional elements among themselves. This was deliberate in order to highlight the capabilities of the language. For instance, the LGCS includes requirements involving the sequencing of property evaluations, which the FCS does not. The FCS, instead, includes requirements involving continuous modulated control through a proportional–integral–derivative loop.

SpecML's reference implementation is dependent on the Eclipse Papyrus modelling environment version that was available at the time of development. This version does not support the targeted versions of SysML and MARTE. However, this by no means limits SpecML's reference implementation since adapters were developed to implement the entire SpecML profile. Selection bias of participants in the workshop is a noteworthy factor to SpecML's internal validity. However, practitioners knowledgeable about the challenges of requirements specification, requirements-based testing and certification were selected.

5.6.2 External validity

Threats to external validity have to do with the generalizability of the results to other design scenarios and systems. The *checsdm* approach was instantiated for two specific design sce-

narios. One of them involving complete overlap between design models created with a mix of UML, Simulink and Stateflow. The other one involving a partial overlap between design models created with a mix of AADL, Simulink and Stateflow. These instantiations resulted in *checsdm4uss*, which was then applied over three avionics systems, and *checsdm4a/ss*. The results might not completely generalize further. However, it is to be remarked that *checsdm*'s *elicitation* phase is intended to define the requirements of any design scenario in question. Hence, *checsdm* should be applicable to any design scenario regardless of domain and modelling languages involved. Moreover, *checsdm4uss* and *checsdm4a/ss* should be applicable as well to other design models that meet their targeted design scenarios. It is worth mentioning that *checsdm4uss* and *checsdm4a/ss*' mapping rules and design guidelines are not closed sets and, thus, more mapping rules and design guidelines could be added since the instantiation of the *checsdm* approach is an iterative process.

SpecML was applied for the requirements specification modelling of two avionics software and their results might not completely generalize further. It is to be remarked that SpecML is intended to capture requirements of avionics software in accordance with DO-178C. However, the language should be applicable to other domains as well where applicable regulations define constraints over requirements specifications as restrictive as those included in SpecML. It is worth mentioning that the set of SpecML constructs is not a closed set and, thus, more stereotypes and constraints could be added since the methodology followed for building SpecML is an iterative process.

Finally, the avionics systems used were already discussed in other existing studies and, thus, are openly-available to allow comparisons and replication. These systems were also considered by the industry partners to be archetypal of their own manufactured systems. These partners have ample experience in domains were safety-critical systems operate and deal with the challenges of requirements specifications, requirements-based testing, heterogeneous design and certification as part of their work. The goal of this evaluation is not to prove our proposed approach applies to all safety-critical systems and domains but rather show its application is feasible and can lead to regulation-compliant requirements specifications as well as to effec-

tive requirement analyses and testing. The evaluation results suggest our proposed approach is likely to provide engineers with potential gains in these regards.

5.7 Chapter Summary

This chapter presented an evaluation of *checsdm*, its two instantiations and SpecML about their feasibility and benefits. The evaluation shows the efforts involved in the applications of *checsdm* were acceptable. In the case of *checsdm4uss*, inconsistencies were identified in the design models of three avionics systems with greater recall rates over fully manual verifications carried out by engineers. The feasibility argument for SpecML was based on the requirements specification of two software-intensive safety-critical avionics systems, one of which was the LGCS presented in Chapter 2. Furthermore, the evaluation also included a workshop conducted with practitioners from the industry partners involved. The goal of the workshop was to assess the likelihood of the proposed approach's adoption in a real industrial context. When presented with the proposed approach, the practitioners perceived it to be advantageous to use within the context of their work and gave an overall likelihood of adoption. The results from this workshop give a good indication of the usefulness and relevance of the approach. The chapter ended with a discussion of the potential threats to the validity of this evaluation.

CONCLUSION AND OUTLOOK

The salient challenges addressed in this thesis arose from the interest of two industry partners to take advantage of MDE to support their efforts of compliance with DO-178C. The avionics systems they manufacture are complex systems combining physical and mechanical components, networking and software. In order to design such systems they deal with 1) diverse components, each one with its own underlying theories and domain vocabularies, and 2) various aspects of the same component, such as their function, structure and behaviour. There are no systematic solutions in the literature for ensuring consistency between heterogeneous design models that cover a wide range of design scenarios and tackle the issue of verifying conformance to design standards. Furthermore, existing requirements specification languages proposed for safety-critical systems development either only support the specification of natural language-based requirements or force requirements to be captured in an already structured form. On top of that, none of these languages provides sufficient support for achieving DO-178C objectives and activities.

In view of these limitations, we pursued the goal of providing an approach, built on the progress made in MDE technologies, to support the development and certification of DO-178C-compliant safety-critical avionics software. In particular, we proposed *checsdm*, a systematic and automated method for assisting engineering teams in ensuring consistency of heterogeneous design models of safety-critical systems. *checsdm* is devised as a *hybrid* approach where automated tools aid engineering teams in flagging errors for review and eventual correction, only to be followed by a more traditional verification process. As a constituting element of *checsdm* we provide SpecML, a modelling language that features a requirements specification infrastructure for safety-critical avionics software development and certification. The following section synthesizes our theoretical and practical contributions.

Contributions

Three features of *checsdm* can be highlighted. First, it aims to cover more design scenarios than existing related approaches by providing a design-scenario-independent framework to support verification of heterogeneous design models. Second, it provides facilities that ease the verification of model consistency and adherence to design standards. Third, it enforces specific aspects of DO-178C compliance needs (*e.g.*, explicitly defining a design scenario and design guidelines, ensuring design consistency) in an effort to aid the recollection of evidence for certification.

checsdm is developed as a generic *methodology* and a *tool framework*. The methodology comprises an iterative three-phased process. The first phase, *elicitation*, consists on eliciting requirements of the heterogeneous design scenario. Three outcomes are expected from the elicitation: a characterization of the mix of modelling languages, a set of mapping rules and a set of (intra- and inter-)design guidelines. The mix of modelling languages were characterized in four dimensions (*i.e.* coverage of system elements, design perspective, level of abstraction and overlap of system elements), which enable coverage of the range of design scenarios. The mapping rules represent the requirements on the relationships between overlapping elements in the different models. The guidelines are intended to help engineers ensure consistency of the design models, while working independently. The second phase, *codification*, consists on deriving a particular toolchain, using the proposed tool framework, that helps engineers in flagging consistency errors for review and eventual correction. Mappings between overlapping model elements are recorded in a mapping model, also capable of flagging consistency issues.

The third phase, *operation*, applies the derived toolchain to actual system designs. Feedback from this phase back to the previous phases can help refine the mapping rules and guidelines or identify new ones. The first step of the *operation* phase relies on SpecML for the requirements specification. SpecML offers a blended modelling approach by integrating constructs

from several existing languages. Three features can be highlighted: 1) it enforces required information (*e.g.*, trace data, decomposition of requirements) for achieving objectives and activities defined in DO-178C and the DO-331 and DO-332 supplements, 2) it captures requirements in natural language to smooth the way for its adoption in industry, and 3) it provides facilities to capture requirements in a structured and semantically-rich formalism to enable requirements-based analyses and testing. Heterogeneous design models can be created from a SpecML requirements specification and, afterwards, automatically verified for compliance with design standards and for consistency.

As for our practical contributions, we presented two instantiations of the *checsdm* approach for the specific cases of a design scenario involving a mix of UML, Simulink and Stateflow (labeled *checsdm4uss*), and a design scenario involving a mix of AADL, Simulink and Stateflow (labeled *checsdm4a/ss*). These design scenarios were motivated in response to industry partners' needs for ensuring consistency between UML, Simulink, Stateflow, and AADL design models for avionics systems that must comply with DO-178C and DO-331. Originating from the *codification* phase of *checsdm4uss*, is *Breesse*. *Breesse* delivers a bridge for the Eclipse Modeling Framework ecosystem and the MathWorks Simulink and Stateflow ecosystem. Furthermore, we developed a reference implementation for SpecML on top of the Eclipse Papyrus modelling environment, although the language itself is tool-independent and any UML modelling tool supporting UML profiles could be used to implement it.

An evaluation of the execution of *checsdm* to derive *checsdm4uss* and *checsdm4a/ss* showed the feasibility and benefits of applying the proposed approach. Furthermore, the evaluation showed the efforts involved in the applications of *checsdm* were acceptable. In the case of *checsdm4uss*, inconsistencies were identified in the design models of three safety-critical avionics systems with greater recall rates over fully manual verifications. We applied SpecML for the requirements specification of two of those safety-critical avionics systems. This evaluation

showed the potential benefits of SpecML for DO-178C- and DO-331-compliant requirements specification. All the requirements of both systems were modelled and, in addition, checked for compliance with DO-178C objectives. We also conducted a workshop with practitioners from the industry partners to assess the proposed approach's likelihood of adoption in a real industrial context. When presented with the approach, the practitioners perceived it to be advantageous to use within the context of their work and gave an overall likelihood of adoption. The results from this workshop gave a good indication of the usefulness and relevance of the approach.

Note that different elements of this thesis have been published. Listed in chronological order, the publications are as follows:

- The modelling of DO-178C was developed with the help of a masters student and the results are detailed in his masters thesis (Metayer, 2018). The modelling of DO-178C was also presented at the 9th International IEEE Workshop on Software Certification (WoSoCer 2019) hosted at the 30th International Symposium on Software Reliability Engineering (ISSRE 2019) (Metayer *et al.*, 2019).

- A characterization of model-based support for DO-178C-compliant avionics software development and certification was presented at the 6th International IEEE Workshop on Software Certification (WoSoCer 2016) hosted at the 27th International Symposium on Software Reliability Engineering (ISSRE 2016) (Paz & El Boussaidi, 2016).

- The LGCS was presented at the 33rd ACM Symposium on Applied Computing (SAC 2018) (Paz & El Boussaidi, 2018). The complete artifacts for the LGCS have been made available online in a technical report (Paz & El Boussaidi, 2017).

- SpecML was presented at the 6th International Workshop on Requirements Engineering and Testing (RET 2019) hosted at the 41st ACM/IEEE International Conference on Software Engineering (ICSE 2019) (Paz & El Boussaidi, 2019b).

- *checsdm4uss* was presented at the 43rd IEEE Annual Computer Software and Applications Conference (COMPSAC 2019) (Paz & El Boussaidi, 2019c).

- *checsdm* has been submitted and accepted with revisions as a manuscript for the journal IEEE Transactions on Software Engineering (IEEE TSE) (Paz *et al.*, 2020).

Limitations and Future Work

We close this thesis by discussing some limitations and logical directions in which this work can be continued and extended.

Evaluation. We established our feasibility and effectiveness arguments on a minimal set of experiments that we could carry out with the available time and participants. We acknowledge the need to expand our experimentation especially in real industrial settings. In fact, we are already working on this front. The proposed approach, including both SpecML and *checsdm* (with its two concrete instantiations *checsdm4uss* and *checsdm4a/ss*), is currently deployed at the industry partners' premises. As part of this, its use on large scale industry systems should provide new input data for evaluation and further refinement and improvement.

LGCS specification and design. The requirements specification we built for the LGCS was defined on the basis of the case study from Boniol & Wiels (2014) and certainly does not cover all kinds of requirements. Furthermore, the LGCS documentation covers only part of the DO-178C software development life cycle, namely requirements specification and design. In the short to medium term, the LGCS requirements specification can be enriched with more requirements, for instance, requirements regarding redundant software operation and management. In

the medium to long term, the documentation can be extended to include software verification and validation artifacts, as required by DO-178C, like verification cases derived from the specified HLRs and the results from their executions against both the design and HLRs. Verification cases can be developed to target normal-range and robustness conditions, as well as coverage of all requirements.

checsdm, checsdm4uss and checsdm4a/ss. The tool framework is considered a prototype. In the short term, improvements can be made to the mapping model and its editor for a more interactive inspection and editing of mappings and consistency issues. In the medium and long terms there are several lines of work that should be pursued.

- While formalizing the guidelines and experimenting with the approach on the different case studies, we found that certain design guidelines can be related to choices in the design process (*e.g.*, av_us_01) and, thus, may be difficult to codify. Additional exploration of this kind of design guidelines should be undertaken to better understand them and devise ways of codifying them.

- Even though DO-178C is considered to be among the group of prescriptive standards, it does leave some open elements that have to be defined in the internal working procedures of each specific company. The studies that will be carried out at our industry partners' premises as part of our approach's deployment should shed light on the consideration of design guidelines related to processes.

- Methodologies already defining ways of using selected modelling languages in the context of embedded control software might exist. These methodologies should be studied and their consideration in *checsdm* explored.

- The introduction of other heuristics into the current instantiations that can further reduce the number of false inconsistencies has been discussed. It is, however, worth investigating their integration into the *codification* phase so that other instantiations may benefit as well.

SpecML. SpecML's reference implementation is considered a prototype. Improvements can be made, in the short term, to its modelling tooling to provide a more interactive inspection and editing of requirements. The industry partners have expressed their desire to have additional functionality explored and developed. We devise two additional functionalities for the medium term: 1) the (semi-)automatic generation of reports from the data that is captured, and 2) the import of existing requirements from other sources (*e.g.*, Microsoft Office Word, Microsoft Office Excel, IBM DOORS). The former is necessary to facilitate the collection of evidence for certification and the latter to enable SpecML's use in ongoing projects without having to completely create the requirements specification models from scratch. We devise to other new functionalities for the long term: 1) the (semi-)automatic translation of the natural language requirement statements into property-based statements, and 2) the interface with existing test generation tools. The former can alleviate the burden placed on engineers when using SpecML to formalize natural language requirements. The latter is intended to satisfy other DO-178C objectives that were not part of our scope for this thesis.

Generalizability. Our proposed approach might not be general enough for its application in other safety-critical domains and certification standards. We expect some degree of applicability in a broader scope since the issue of heterogeneity occurs as well for other types of cyber-physical systems and the DO-178C guideline gathers common best development and safety practices that have been followed to produce safe systems. However, since we solely focused on DO-178C and its accompanying supplements, our proposed approach and results are really based on such analysis. Further work is necessary to analyze other standards, guidelines and norms for certification in different domains (*e.g.*, IEC 61508, IEC 60880, ISO 26262,

CENELEC EN 50126) and evaluate *checsdm*'s feasibility and effectiveness for them. Results from such analysis and evaluation should reveal what other extensions must be done at the theoretical and practical levels to adapt and extend our proposed approach accordingly.

APPENDIX I

THE CHARACTERIZATION FRAMEWORK

We developed a framework to characterize and compare model-based approaches according with: 1) their objectives and targeted stakeholders, 2) the extent to which they support DO-178C guidelines, 3) the way they handle and present information, and 4) the extent to which the approach is ready for use. Thus, as shown in Figure-A I-1, our framework defines four dimensions named: *philosophy*, *DO-178C coverage*, *information handling* and *usage*. The following sections present each of these dimensions and the criteria that fall under them.

1. Philosophy

The *philosophy* of an approach can be divided into *objective*, *object* of interest, and *targeted stakeholder group*. An approach's *objective* collects its main aims and provides an indication of the vision upon which it has been established. Objectives are grouped into three main categories: 1) *specify*, 2) *analyze*, and 3) *generate*. *Specify* objectives indicate the intention of capturing information describing elements that pertain to an application domain in order to allow its understanding, comprehension or communication. Included under the *analyze* category are those aims addressing the satisfaction of properties, *e.g.*, model verification. Include under the *generate* category are the objectives of transforming models into other work artifacts, be these other models or text.

The *object(s)* of interest of an approach gives a clear identification of its modeling interests. We outline as objects of interest: 1) *requirements*, 2) *architecture*, 3) *processes*, 4) *safety information* (*e.g.*, hazards, events and responses), 5) *tests*, 6) *regulation*, and 7) *reports*. The previous listing is not exhaustive and could incorporate more options as needed. The *targeted stakeholder group* criteria captures the descriptions of the stakeholders taken into account by the modeling approaches, which can be *system*, *software* or *safety*.

2. DO-178C Coverage

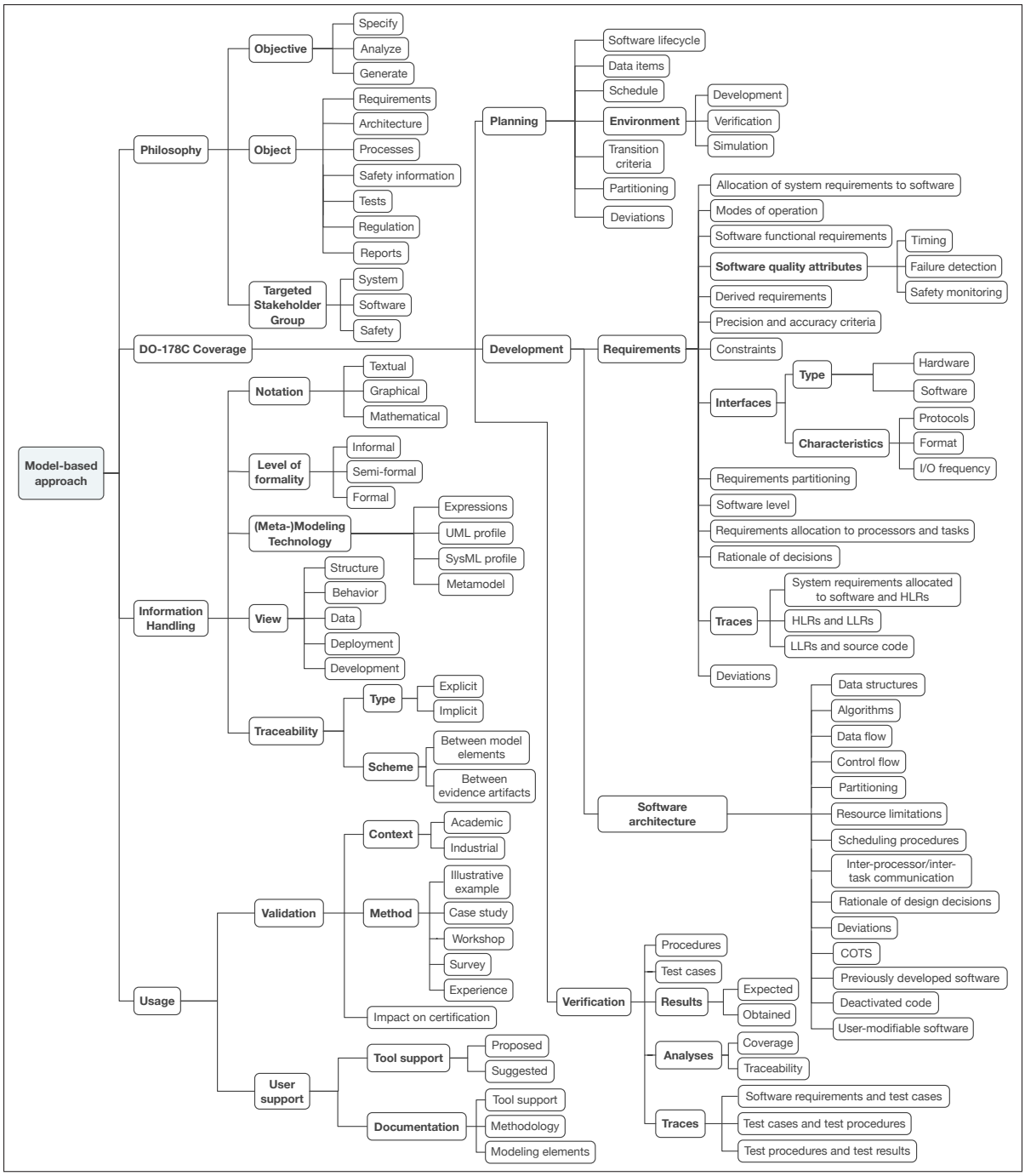


Figure-A I-1 The characterization framework. Extracted from Paz & El Boussaidi (2016).

The *DO-178C coverage* dimension provides a description of the coverage extent for DO-178C's software life cycle data item contents to which the modeling approach is relevant. We

only consider the contents of the data items from a subset of the software life cycle processes identified in DO-178C, namely, the *planning*, *development* and *verification* processes. We consider incorporating the remaining life cycle processes (*i.e.* configuration management, quality assurance and certification liaison) in a future work.

The *planning* criteria considers modeling support for the *software life cycle* followed, software life cycle data or *data items* that need to be created, development *schedule*, *environments* used (which can be for *development*, *verification* or *simulation*), *transition criteria* between activities to allow moving into other activities while the current activity is not yet complete, and *deviations* from plans.

In the *development* process of DO-178C we explore two key elements: *requirements* (both HLRs and LLRs) and *software architecture*. Within *requirements* we look at assistance for managing the *allocation of system requirements to software*, *modes of operation* (*i.e.* the distinct behaviors of the system; *e.g.*, normal, failed), *software functional requirements*, *software quality attributes* (*e.g.*, timing, failure detection, safety monitoring), *derived requirements*, *precision and accuracy criteria* (*e.g.*, latency, allowed deviations of values from their ideal value), *constraints* (*e.g.*, timing, memory), *interfaces* (with their type: hardware, software; and their characteristics: protocols, format, frequency of inputs and outputs), *requirements partitioning*, design assurance level or *software level*, *requirements allocation to processors and tasks*, *rationale of decisions*, *traces* (between system requirements allocated to software and HLRs, HLRs and LLRs, and LLRs and source code), and *deviations* from requirements. Beneath *software architecture*, the criteria focuses around modeling support for *data structures*, *algorithms*, *data flow*, *control flow*, *partitioning* into components or subsystems, *resource limitations*, *scheduling procedures*, *inter-processor/inter-task communication*, *rationale of design decisions* and *deviations* from design. It also considers support at the design level for the use of *Commercial Off-The-Shelf (COTS) software*, *previously developed software* or *user-modifiable software* and identifying the presence of *deactivated code*. Concerning *verification*, we observe for treatment of verification *procedures* (for reviews, analyses, and testing), *test cases*, verification *results* (expected and obtained), verification *analyses* (coverage and traceability), and *traces* (between

software requirements and test cases, test cases and test procedures, and test procedures and test results).

3. Information Handling

The way modeling approaches handle and present information is an important aspect to consider since it impacts their ability to automate some tasks and their usability. In this regard, we are taking into account an approach's modeling method or *notation*, which gives a description of the way elements are represented by it (*i.e. textual*, *graphical* or *mathematical*), and their *level of formality* (*informal*, *semi-formal* or *formal*). Likewise, we look into the *(meta-)modeling technology* the approach is built with, be it *UML profile*, *SysML profile*, *metamodel* or mathematical *expressions*. We also evaluate the approaches' provided *views* for a description of the aspects that can be represented by the modeling approaches. Views are grouped into five categories: 1) *structure* (a topological description of the modeling elements, *i.e.* the structural dependencies between the modeling elements), 2) *behavior* (sequencing or control of information among the modeling elements), 3) *development* (high-level structural arrangement of the modeling elements) 4) *deployment* (addresses the descriptions of who performs a task, function or activity and where), and 5) *data* (structure and dependencies between the data entities produced or manipulated). In addition, we inquire about active *traceability* support since it allows the tracking of evidence artifacts and their contents across the life cycle. We recognize two *types* of traceability links: explicit (or direct) and implicit (or transitive). We also examine the *scheme* in which these traces are captured, *i.e.* either *between modeled elements* or *between evidence artifacts* or both.

3.1 Usage

Usage of an approach is a desirable characteristic, yet it is not always possible because it lacks validation or proper documentation. *Validation* consists of a description of the *context* or setting (*academic* or *industrial*), the employed methods for any of the validation activities (*example*, *case study*, *workshop*, *survey* or *experience*), and an indication of any supporting evidence

that the modeling approach led to or had any *impact on certification*. *User support* involves descriptions of the support material and tools for using the modeling approaches. Criteria for user support consist of *tool support*, *i.e.* technology, tools and techniques *proposed* or *suggested* for use, and *documentation* of practical guidance, which may include documentation of *tool support*, *methodology* or *modeling elements*.

APPENDIX II

LANDING GEAR CONTROL SOFTWARE REQUIREMENTS SPECIFICATION AND DESIGN (BASELINE)

MV-1 *Desired Gear Position*. The LGCS shall process the `Desired Gear Position` monitorable variable from the pilot interface.

Name of variable	`Desired Gear Position`
Type	Enumeration
Expected Range/Values	Up, ⊛Down
Interpretation	Command to extend or retract the gears: • Up: Retract gears. • Down: Extend gears.
Precision	n/a
Units	n/a
Load	One-time message on pilot input.
Exception Handling Information	When no pilot input is given, value is treated as null.

⊛ denotes initial value

Rationale: This is the specified input received from the pilot interface given the system's operational context.

MV-2 *Analogical Switch Status*. The LGCS shall process three `Analogical Switch Status` monitorable variables from the analogical switch sensor.

Name of variable	`Analogical Switch Status`
Type	Enumeration
Expected Range/Values	⊛Open, Closed
Interpretation	Status of the analogical switch.
Precision	n/a
Units	n/a
Load	1000 Hz signal
Exception Handling Information	n/a

⊛ denotes initial value

Rationale: This is the specified input received from the analogical switch sensor given the system's operational context.

MV-3 *Hydraulic Circuit Pressure*. The LGCS shall process three `Hydraulic Circuit Pressure` monitorable variables from the hydraulic circuit pressure sensor.

Name of variable	Hydraulic Circuit Pressure
Type	Real
Expected Range/Values	[0.0 .. 40,000.0]
Interpretation	Current pressure in the hydraulic circuit.
Precision	± 0.1
Units	kPa
Load	1000 Hz signal
Exception Handling Information	Out of range values are treated as -1.

Rationale: This is the specified input received from the hydraulic pressure sensor given the system's operational context.

MV-4 *Door Close Status*. The LGCS shall process three Door Close Status monitorable variables from the door closed sensors in the nose and main wheel assemblies.

Name of variable	Door Closed Status
Type	Enumeration
Expected Range/Values	Not Closed, ⊛Closed
Interpretation	Status of the doors: • Not Closed: Door is not completely closed. • Closed: Door is completely closed.
Precision	n/a
Units	n/a
Load	1000 Hz signal
Exception Handling Information	n/a

⊛ denotes initial value

Rationale: This is the specified input received from the door closed sensors given the system's operational context.

MV-5 *Door Open Status*. The LGCS shall process three Door Open Status monitorable variables from the door open sensors in the nose and main wheel assemblies.

Name of variable	Door Open Status
Type	Enumeration
Expected Range/Values	⊛Not Open, Open
Interpretation	Status of the doors: • Not Open: Door is not completely open. • Open: Door is completely open.
Precision	n/a
Units	n/a
Load	1000 Hz signal
Exception Handling Information	n/a

⊛ denotes initial value

Rationale: This is the specified input received from the door open sensors in the nose and main wheel assemblies given the system's operational context.

MV-6 *Gear Retracted Status*. The LGCS shall process three `Gear Retracted Status` monitorable variables from the gear retracted sensors in the nose and main wheel assemblies.

Name of variable	`Gear Retracted Status`
Type	Enumeration
Expected Range/Values	⊛Not Retracted, Retracted
Interpretation	Status of the gears: • Not retracted: Gear is not fully retracted. • Retracted: Gear is fully retracted.
Precision	n/a
Units	n/a
Load	1000 Hz signal
Exception Handling Information	n/a

⊛ denotes initial value

Rationale: This is the specified input received from the gear retracted sensors in the nose and main wheel assemblies given the system's operational context.

MV-7 *Gear Extended Status*. The LGCS shall process three `Gear Extended Status` monitorable variables from the gear extended sensors in the nose and main wheel assemblies.

Name of variable	`Gear Extended Status`
Type	Enumeration
Expected Range/Values	Not Extended, ⊛Extended
Interpretation	Status of the gears: • Not extended: Gear is not fully extended. • Extended: Gear is fully extended.
Precision	n/a
Units	n/a
Load	1000 Hz signal
Exception Handling Information	n/a

⊛ denotes initial value

Rationale: This is the specified input received from the gear extended sensors in the nose and main wheel assemblies given the system's operational context.

MV-8 *Gear Shock Absorber Status*. The LGCS shall process three `Gear Shock Absorber Status` monitorable variables from the gear shock absorber sensors in the nose and main wheel assemblies.

Name of variable	`Gear Shock Absorber Status`
Type	Enumeration
Expected Range/Values	⊛Unrelaxed, Relaxed
Interpretation	Status of the gears' shock absorbers: • Unrelaxed: Shock absorber is not relaxed. • Relaxed: Shock absorber is relaxed.
Precision	n/a
Units	n/a
Load	1000 Hz signal
Exception Handling Information	n/a

⊛ denotes initial value

Rationale: This is the specified input received from the gear shock absorber sensors in the nose and main wheel assemblies given the system's operational context.

CV-1 *General Electro-Valve*. The LGCS shall output actuation commands to the general hydraulic electro-valve through the `General EV Actuation Command` controllable variable.

Name of variable	`General EV Actuation Command`
Type	Enumeration
Expected Range/Values	⊛Close, Open
Interpretation	Command to actuate the general electro-valve.
Precision	n/a
Units	n/a
Load	One-time message
Exception Handling Information	n/a

⊛ denotes initial value

Rationale: This is the specified output sent to the general electro-valve given the system's operational context.

CV-2 *Door Closing Electro-Valve*. The LGCS shall output actuation commands to the door closing hydraulic electro-valve through the `Door Closing EV Actuation Command` controllable variable.

Name of variable	Door Closing EV Actuation Command
Type	Enumeration
Expected Range/Values	⊛Close, Open
Interpretation	Command to actuate the door closing electro-valve.
Precision	n/a
Units	n/a
Load	One-time message
Exception Handling Information	n/a

⊛ denotes initial value

Rationale: This is the specified output sent to the door closing electro-valve given the system's operational context.

CV-3 *Door Opening Electro-Valve*. The LGCS shall output actuation commands to the door opening hydraulic electro-valve through the Door Opening EV Actuation Command controllable variable.

Name of variable	Door Opening EV Actuation Command
Type	Enumeration
Expected Range/Values	⊛Close, Open
Interpretation	Command to actuate the door opening electro-valve.
Precision	n/a
Units	n/a
Load	One-time message
Exception Handling Information	n/a

⊛ denotes initial value

Rationale: This is the specified output sent to the door opening electro-valve given the system's operational context.

CV-4 *Gear Retraction Electro-Valve*. The LGCS shall output actuation commands to the gear retraction hydraulic electro-valve through the Gear Retraction EV Actuation Command controllable variable.

Rationale: This is the specified output sent to the gear retraction electro-valve given the system's operational context.

Name of variable	Gear Retraction EV Actuation Command
Type	Enumeration
Expected Range/Values	⊛Close, Open
Interpretation	Command to actuate the gear retraction electro-valve.
Precision	n/a
Units	n/a
Load	One-time message
Exception Handling Information	n/a

⊛ denotes initial value

CV-5 *Gear Extension Electro-Valve*. The LGCS shall output actuation commands to the gear extension hydraulic electro-valve through the Gear Extension EV Actuation Command controllable variable.

Name of variable	Gear Extension EV Actuation Command
Type	Enumeration
Expected Range/Values	⊛Close, Open
Interpretation	Command to actuate the gear extension electro-valve.
Precision	n/a
Units	n/a
Load	One-time message
Exception Handling Information	n/a

⊛ denotes initial value

Rationale: This is the specified output sent to the gear extension electro-valve given the system's operational context.

CV-6 *Feedback*. The LGCS shall output feedback to the pilot interface through the Feedback controllable variable.

Rationale: This is the specified output sent to the pilot interface given the system's operational context.

SRATS-1 *Sensor Validation*. When the LGCS receives data from one of the LGS sensors, the LGCS shall process the three readings associated to the sensor data based on the following rules:

a. If the three readings are indicating valid values and are equal, then the overall sensor value is this common value and is considered valid.

Name of variable	`Feedback`
Type	Enumeration
Expected Range/Values	Off, Amber, ⊛Green, Red
Interpretation	Indication of the gears' current position and the state of the LGCS: • Off: LGCS is functioning and gears are retracted. • Amber: LGCS is functioning and gears are in transit to desired position. • Green: LGCS is functioning and gears are extended. • Red: LGCS has failed.
Precision	n/a
Units	n/a
Load	One-time message
Exception Handling Information	n/a

⊛ denotes initial value

b. If the three readings are indicating valid values but are different, then the readings are considered invalid. The overall sensor value is any of the readings' value and is considered invalid.

c. If the three readings are indicating valid values but one reading is different from the other two for the first time, then that reading is considered invalid and is permanently eliminated, i.e. only the two remaining valid readings are considered in the future. The overall sensor value is the common value of the two remaining valid readings and is considered valid.

d. If the two remaining readings are indicating valid values but are different, then these readings are considered invalid. The overall sensor value is any of the reading's value and is considered invalid.

e. If one of the readings is indicating an invalid value and the other two readings are indicating valid values that are equal, then that reading is permanently eliminated, i.e. only the two remaining valid readings are considered in the future. The overall sensor value is the common value of the two remaining valid readings and is considered valid.

f. If at least two readings are indicating invalid values, then the overall sensor value is any of the reading's value and is considered invalid.

g. If the overall sensor value is invalid, then a failure of such sensor shall be detected.

Rationale: A sensor should not be trusted if it cannot accurately describe the status of its associated system entity. For redundancy purposes, all the sensors perform three independent readings describing simultaneously the same situation. Thus, a voting strategy is followed to determine the overall value and validity of the sensor. At least two of the three readings must yield an equal value.

Traceability: Developed into HLR-1.

SRATS-2 *Retraction Sequence*. When an Up Desired Gear Position is received, the LGCS shall carry out the following sequence of actions in under 28 seconds:

1. Open the General EV.
2. Open the Door Opening EV.
3. When the Door Open Status for all wheel assemblies is Open and the Door Closed Status for all wheel assemblies is Not Closed, close the Door Opening EV.
4. Turn Amber the feedback indicator on the pilot interface.
5. If the Gear Shock Absorber status for all wheel assemblies is Relaxed, Open the Gear Retraction EV. Else, turn Green the feedback indicator on the pilot interface and skip to 8.
6. When the Gear Retracted Status for all wheel assemblies is Retracted and the Gear Extended Status for all wheel assemblies is Not Extended, close the Gear Retraction EV.
7. Turn Off the feedback indicator on the pilot interface.
8. Open the Door Closing EV.
9. When the Door Closed Status for all wheel assemblies is Closed and the Door Opening Status for all wheel assemblies is Not Open, close the Door Closing EV.

10. Close the General EV.

Rationale: The system should not perform any functions unless a desired gear position is received. This is the specified gear retraction sequence to be performed given the system's mechanical parts and hydraulic circuit elements. The maximum time for a gear retraction sequence to be completed is 28 seconds after the gear lever has changed position.

Traceability: Developed into HLR-2.

SRATS-3 *Extension Sequence*. When a Down Desired Gear Position is received, the LGCS shall carry out the following sequence of actions in under 28 seconds:

1. Open the General EV.
2. Open the Door Opening EV.
3. When the Door Open Status for all wheel assemblies is Open and the Door Closed Status for all wheel assemblies is Not Closed, close the Door Opening EV.
4. Turn Amber the feedback indicator on the pilot interface.
5. Open the Gear Extension EV.
6. When the Gear Extended Status for all wheel assemblies is Extended and the Gear Retracted Status for all wheel assemblies is Not Retracted, close the Gear Extension EV.
7. Turn Green the feedback indicator on the pilot interface.
8. Open the Door Closing EV.
9. When the Door Closed Status for all wheel assemblies is Closed and the Door Opening Status for all wheel assemblies is Not Open, close the Door Closing EV.
10. Close the General EV.

Rationale: The system should not perform any functions unless a desired gear position is received. This is the specified gear extension sequence to be performed given the system's mechanical parts and hydraulic circuit elements. The maximum time for a gear extension sequence to be completed is 28 seconds after the gear lever has changed position.

Traceability: Developed into HLR-3.

SRATS-4 *Cancel Sequence*. When the LGCS is currently executing a retraction sequence and a Down Desired Gear Position is received, the LGCS shall halt the current retraction sequence and revert all the actions that were executed. Conversely, when the LGCS is currently executing an extension sequence and an Up Desired Gear Position is received, the LGCS shall halt the current extension sequence and revert all the actions that were executed.

Rationale: A gear motion can be canceled by the pilot or copilot at any step of an ongoing sequence. Any action executed of the ongoing sequence has to be reverted.

Traceability: Developed into HLR-4.

SRATS-5 *General EV Actuation*. When a Desired Gear Position is received, the LGCS shall wait for up to 1 second for the Analogical Switch to be Closed to be able to Open the General EV. Once 1 second has elapsed since a Desired Gear Position was received and the Analogical Switch is still Open, the LGCS shall detect a failure of the analogical switch.

Rationale: This is the specified operation to protect the hydraulic circuit from being operational without the pilot's command. The analogical switch should be closed after 0.8 seconds after the landing gear lever has changed positions. The analogical switch being open after this time is an indication of its failure. The additional 0.2 seconds provides a hysteresis that prevents transient alarms.

Traceability: Developed into HLR-5.

SRATS-6 *Hydraulic Circuit Pressure for Specific EV Actuation*. Once the Hydraulic Circuit Pressure is greater than or equal to 30,000 kPa and less than 35,000 kPa after the General EV is Opened, the LGCS can Open the necessary specific EV (*i.e.* Door Opening EV, Door Closing EV, Gear Retraction EV or Gear Extension EV).

Rationale: This is the specified oil pressure for operating the specific electro-valves.

Traceability: Developed into HLR-6.

SRATS-7 *Consecutive EV Actuation Delay*. Once at least 0.2 seconds have elapsed since the General EV was Opened, the LGCS can Open the necessary specific EV (*i.e.* Door Opening EV, Door Closing EV, Gear Retraction EV or Gear Extension EV).

Rationale: This is the specified minimum wait time between stimulations of the general electro-valve and the specific electro-valves due to oil pressure differential produced by the change in fluid velocity across the hydraulic circuit.

Traceability: Developed into HLR-7.

SRATS-8 *Consecutive Specific EV Actuation Delay*. Once at least 0.2 seconds have elapsed since any of the specific EVs (*i.e.* Door Opening EV, Door Closing EV, Gear Retraction EV or Gear Extension EV) was Closed, the LGCS can Open any other specific EV.

Rationale: This is the specified minimum wait time between stimulations of the specific electro-valves due to oil pressure differential produced by the change in fluid velocity across the hydraulic circuit.

Traceability: Developed into HLR-8.

SRATS-9 *Close General EV Actuation Command*. Once at least 1 second has elapsed since any of the specific EVs (*i.e.* Door Opening EV, Door Closing EV, Gear Retraction EV or Gear Extension EV) was Closed, the LGCS can Close the General EV.

Rationale: This is the specified minimum wait time between stopping stimulations of the specific electro-valves and the general electro-valve due to oil pressure differential produced by the change in fluid velocity across the hydraulic circuit.

Traceability: Developed into HLR-9.

SRATS-10 *Contrary Specific EV Actuation Delay*. Once at least 0.1 seconds have elapsed since any of the specific EVs (*i.e.* Door Opening EV, Door Closing EV, Gear Retraction EV or Gear Extension EV) was Closed, the LGCS can Open its contrary EV (*i.e.* Door Closing EV Actuation Command / Door Opening EV Actuation Command, or Gear Extension EV Actuation Command / Gear Retraction EV Actuation Command).

Rationale: This is the specified minimum wait time between contrary stimulations of the specific electro-valves due to oil pressure differential produced by the change in fluid velocity across the hydraulic circuit.

Traceability: Developed into HLR-10.

SRATS-11 *Analogical Switch Closed Failure*. Once 1.5 seconds have elapsed after the 28 second time interval given for the retraction and extension sequences, and no Desired Gear Position has been received and the Analogical Switch is still Closed, the LGCS shall detect a failure of the analogical switch.

Rationale: The analogical switch should be open after 1.2 seconds after the landing gear lever has changed positions. The analogical switch being closed after this time is an indication of its failure. The additional 0.3 seconds provides a hysteresis that prevents transient alarms.

Traceability: Developed into HLR-11.

SRATS-12 *Hydraulic Circuit Unpressurized Failure*. Once 2 seconds have elapsed since the General EV was Opened and the Hydraulic Circuit Pressure is still less than 30,000 kPa, the LGCS shall detect a failure of the general hydraulic electro-valve.

Rationale: The hydraulic circuit should be pressurized in less than 2 seconds after the general electro-valve has been stimulated. The hydraulic circuit being unpressurized after this time is an indication of its failure.

Traceability: Developed into HLR-12.

SRATS-13 *Hydraulic Circuit Pressurized Failure*. Once 10 seconds have elapsed since the General EV was Closed and the Hydraulic Circuit Pressure is still greater than or equal to 30,000 kPa and less than 35,000 kPa, the LGCS shall detect a failure of the general hydraulic electro-valve.

Rationale: The hydraulic circuit should be unpressurized in less than 10 seconds after the general electro-valve has stopped being stimulated. The hydraulic circuit being pressurized after this time is an indication of its failure.

Traceability: Developed into HLR-13.

SRATS-14 *Door Opening Failure*. Once 7 seconds have elapsed since the Door Opening EV was Opened and the Door Closed Status for any wheel assembly is still Closed or the Door Open Status for any wheel assembly is still Not Open, the LGCS shall detect a failure of the doors.

Rationale: The doors should be locked open in less than 7 seconds after stimulating the door opening electro-valve. The doors being closed after this time is an indication of their failure.

Traceability: Developed into HLR-14.

SRATS-15 *Door Closing Failure*. Once 7 seconds have elapsed since the Door Closing EV was Opened and the Door Open Status for any wheel assembly is still Open or the Door Closed Status for any wheel assembly is still Not Closed, the LGCS shall detect a failure of the doors.

Rationale: The doors should be locked closed in less than 7 seconds after stimulating the door closing electro-valve. The doors being open after this time is an indication of their failure.

Traceability: Developed into HLR-15.

SRATS-16 *Gear Retraction Failure*. Once 10 seconds have elapsed since the Gear Retraction EV was Opened and the Gear Extended Status for any wheel assembly is still Extended or the Gear Retracted Status for any wheel assembly is still Not Retracted, the LGCS shall detect a failure of the gears.

Rationale: The gears should be retracted in less than 10 seconds after stimulating the gear retraction electro-valve. The gears being extended after this time is an indication of their failure.

Traceability: Developed into HLR-16.

SRATS-17 *Gear Extension Failure*. Once 10 seconds have elapsed since the Gear Extension EV was Opened and the Gear Retracted Status for any wheel assembly is still Retracted or the Gear Extended Status for any wheel assembly is still Not Extended, the LGCS shall detect a failure of the gears.

Rationale: The gears should be extended in less than 10 seconds after stimulating the gear extension electro-valve. The gears being retracted after this time is an indication of their failure.

Traceability: Developed into HLR-17.

SRATS-18 *Failure Detected.* When the LGCS is currently executing a retraction or extension sequence and a failure is detected, the LGCS shall halt the currently executing sequence and turn Red the feedback indicator on the pilot interface.

Rationale: The system should not perform any functions after a failure is detected.

Traceability: Developed into HLR-18.

HLR-1 *Sensor Validity Check*. When the LGCS receives data from one of the LGS sensors, the LGCS shall process the three monitorable variables associated to the sensor data based on the following rules:

a. If the three monitorable variables are indicating valid values and are equal, then the overall sensor value is this common value and is considered valid.

b. If the three monitorable variables are indicating valid values but are different, then the monitorable variables are considered invalid. The overall sensor value is any of the monitorable variables' value and is considered invalid.

c. If the three monitorable variables are indicating valid values but one monitorable variable is different from the other two for the first time, then that monitorable variable is considered invalid and is permanently eliminated, i.e. only the two remaining valid monitorable variables are considered in the future. The overall sensor value is the common value of the two remaining valid monitorable variables and is considered valid.

d. If the two remaining monitorable variables are indicating valid values but are different, then these monitorable variables are considered invalid. The overall sensor value is any of the monitorable variable's value and is considered invalid.

e. If one of the monitorable variables is indicating an invalid value and the other two monitorable variables are indicating valid values that are equal, then that monitorable variable is permanently eliminated, i.e. only the two remaining valid monitorable variables are considered in the future. The overall sensor value is the common value of the two remaining valid monitorable variables and is considered valid.

f. If at least two monitorable variables are indicating invalid values, then the overall sensor value is any of the monitorable variable's value and is considered invalid.

g. If the overall sensor value is invalid, then a failure of such sensor shall be detected.

Rationale: A sensor should not be trusted if it cannot accurately describe the status of its associated system entity. For redundancy purposes, all the sensors perform three independent readings describing simultaneously the same situation. Thus, a voting

strategy is followed to determine the overall value and validity of the sensor. At least two of the three readings must yield an equal value. The use of the data from a sensor is dependent on a prior validity check to avoid CFC-2.

Traceability: Developed from SRATS-1. Developed into LLR-1, LLR-2, LLR-3, LLR-4, LLR-5, LLR-6, LLR-7, LLR-8, LLR-9 and LLR-10.

HLR-2 *Retraction Sequence Control*. When an Up value is received for the Desired Gear Position monitorable variable, the LGCS shall carry out the following sequence of actions in under 28 seconds:

1. Set the General EV Actuation Command controllable variable to Open.
2. Set the Door Opening EV Actuation Command controllable variable to Open.
3. When the overall value of the Door Open Status monitorable variables for all wheel assemblies is Open and the overall value of the Door Closed Status monitorable variables for all wheel assemblies is Not Closed, set the Door Opening EV Actuation Command controllable variable to Close.
4. Set the Feedback controllable variable to Amber.
5. If the overall value of the Gear Shock Absorber Status monitorable variables for all wheel assemblies is Relaxed, set the Gear Retraction EV Actuation Command controllable variable to Open. Else, set the Feedback controllable variable to Green and skip to 8.
6. When the overall value of the Gear Retracted Status monitorable variables for all wheel assemblies is Retracted and the overall value of the Gear Extended Status monitorable variables for all wheel assemblies is Not Extended, set the Gear Retraction EV Actuation Command controllable variable to Close.
7. Set the Feedback controllable variable to Off.
8. Set the Door Closing EV Actuation Command controllable variable to Open.

9. When the overall value of the Door Closed Status monitorable variables for all wheel assemblies is Closed and the overall value of the Door Open Status monitorable variables for all wheel assemblies is Not Open, set the Door Closing EV Actuation Command controllable variable to Close.
10. Set the General EV Actuation Command controllable variable to Close.

Rationale: The system should not perform any functions unless a desired gear position is received. This is the specified gear retraction sequence to be performed given the system's mechanical parts and hydraulic circuit elements. The maximum time for a gear retraction sequence to be completed is 28 seconds after the gear lever has changed position. If any of the gear shock absorbers is not relaxed it is an indication that the aircraft is on the ground. The aircraft should maintain the gears extended when it is on the ground to avoid CFC-1.

Traceability: Developed from SRATS-2. Developed into LLR-11, LLR-12, LLR-13, LLR-14, LLR-15, LLR-16, LLR-17, LLR-18, LLR-19, LLR-20, LLR-21, LLR-22, LLR-23, LLR-24, LLR-25, LLR-26, LLR-27 and LLR-40.

HLR-3 *Extension Sequence Control*. When a Down value is received for the Desired Gear Position monitorable variable, the LGCS shall carry out the following sequence of actions in under 28 seconds:

1. Set the General EV Actuation Command controllable variable to Open.
2. Set the Door Opening EV Actuation Command controllable variable to Open.
3. When the overall value of the Door Open Status monitorable variables for all wheel assemblies is Open and the overall value of the Door Closed Status monitorable variables for all wheel assemblies is Not Closed, set the Door Opening EV Actuation Command controllable variable to Close.
4. Set the Feedback controllable variable to Amber.

5. Set the Gear Extension EV Actuation Command controllable variable to Open.
6. When the overall value of the Gear Extended Status monitorable variables for all wheel assemblies is Extended and the overall value of the Gear Retracted Status monitorable variables for all wheel assemblies is Not Retracted, set the Gear Extension EV Actuation Command controllable variable to Close.
7. Set the Feedback controllable variable to Green.
8. Set the Door Closing EV Actuation Command controllable variable to Open.
9. When the overall value of the Door Closed Status monitorable variables for all wheel assemblies is Closed and the overall value of the Door Open Status monitorable variables for all wheel assemblies is Not Open, set the Door Closing EV Actuation Command controllable variable to Close.
10. Set the General EV Actuation Command controllable variable to Close.

Rationale: The system should not perform any functions unless a desired gear position is received. This is the specified gear extension sequence to be performed given the system's mechanical parts and hydraulic circuit elements. The maximum time for a gear extension sequence to be completed is 28 seconds after the gear lever has changed position.

Traceability: Developed from SRATS-3. Developed into LLR-11, LLR-12, LLR-13, LLR-14, LLR-15, LLR-16, LLR-25, LLR-26, LLR-27, LLR-28, LLR-29, LLR-30, LLR-31, LLR-32, LLR-33 and LLR-40.

HLR-4 *Cancel Sequence Control*. When the LGCS is currently executing a retraction sequence and a Down value is received for the Desired Gear Position monitorable variable, the LGCS shall halt the current retraction sequence and revert all the actions that were executed. Conversely, when the LGCS is currently executing an extension sequence and an Up value is received for the Desired Gear Position monitorable

variable, the LGCS shall halt the current extension sequence and revert all the actions that were executed.

Rationale: A gear motion can be canceled by the pilot or copilot at any step of an ongoing sequence. Any action executed of the ongoing sequence has to be reverted.

Traceability: Developed from SRATS-4. Developed into LLR-21, LLR-32, LLR-34, LLR-35, LLR-36 and LLR-37.

HLR-5 *General EV Actuation Control*. When a Desired Gear Position is received, the LGCS shall wait for up to 1 second for the overall value of the Analogical Switch Status monitorable variables to be Closed to be able to set to Open the General EV Actuation Command controllable variable. Once 1 second has elapsed since a Desired Gear Position was received and the overall value of the Analogical Switch Status monitorable variables is still Open, the LGCS shall detect a failure of the analogical switch.

Rationale: This is the specified operation to protect the hydraulic circuit from being operational without the pilot's command. The analogical switch should be closed after 0.8 seconds after the landing gear lever has changed positions. The analogical switch being open after this time is an indication of its failure. The additional 0.2 seconds provides a hysteresis that prevents transient alarms.

Traceability: Developed from SRATS-5. Developed into LLR-38, LLR-39, LLR-40, LLR-41 and LLR-42.

HLR-6 *Hydraulic Circuit Pressure for Specific EV Actuation Control*. Once the overall value of the Hydraulic Circuit Pressure monitorable variables is greater than or equal to 30,000 kPa and less than 35,000 kPa after the General EV Actuation Command controllable variable is set to Open, the LGCS can set to Open the controllable variable for actuating 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).

Rationale: This is the specified oil pressure needed in the hydraulic circuit for operating the specific electro-valves.

Traceability: Developed from SRATS-6. Developed into LLR-14, LLR-43, LLR-44 and LLR-45.

HLR-7 *Consecutive EV Actuation Delay Control*. Once at least 0.2 seconds have elapsed since the General EV Actuation Command controllable variable was set to Open, the LGCS can set to Open the controllable variable for actuating 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).

Rationale: This is the specified minimum wait time between stimulations of the general electro-valve and the specific electro-valves due to oil pressure differential produced by the change in fluid velocity across the hydraulic circuit.

Traceability: Developed from SRATS-7. Developed into LLR-14 and LLR-46.

HLR-8 *Consecutive Specific EV Actuation Delay Control*. Once at least 0.2 seconds have elapsed since any of the Door Closing EV Actuation Command, Door Opening EV Actuation Command, Gear Retraction EV Actuation Command and Gear Extension EV Actuation Command controllable variables was set to Close, the LGCS can set to Open any other of them.

Rationale: This is the specified minimum wait time between stimulations of the specific electro-valves due to oil pressure differential produced by the change in fluid velocity across the hydraulic circuit.

Traceability: Developed from SRATS-8. Developed into LLR-17, LLR-24, LLR-28, LLR-33 and LLR-47.

HLR-9 *Close General EV Actuation Command Control*. Once at least 1 second has elapsed since any of the Door Closing EV Actuation Command, Door Opening EV Actuation Command, Gear Retraction EV Actuation Command or Gear Exten-

sion EV Actuation Command controllable variables was set to Close, the LGCS can set the General EV Actuation Command controllable variable to Close.

Rationale: This is the specified minimum wait time between stopping stimulations of the specific electro-valves and the general electro-valve due to oil pressure differential produced by the change in fluid velocity across the hydraulic circuit.

Traceability: Developed from SRATS-9. Developed into LLR-27 and LLR-48.

HLR-10 *Contrary Specific EV Actuation Delay Control.* Once at least 0.1 seconds have elapsed since any of the Door Closing EV Actuation Command, Door Opening EV Actuation Command, Gear Retraction EV Actuation Command or Gear Extension EV Actuation Command controllable variables was set to Close, the LGCS can set its contrary controllable variable (*i.e.*Door Closing EV Actuation Command / Door Opening EV Actuation Command, or Gear Extension EV Actuation Command / Gear Retraction EV Actuation Command) to Open.

Rationale: This is the specified minimum wait time between contrary stimulations of the specific electro-valves due to oil pressure differential produced by the change in fluid velocity across the hydraulic circuit.

Traceability: Developed from SRATS-10. Developed into LLR-49 and LLR-50.

HLR-11 *Analogical Switch Close Failure Detection.* Once 1.5 seconds have elapsed after the 28 second time interval given for the retraction and extension sequences and no Desired Gear Position has been received and the overall value of the Analogical Switch Status monitorable variables is still Closed, the LGCS shall detect a failure of the analogical switch.

Rationale: The analogical switch should be open after 1.2 seconds after the landing gear lever has changed positions. The analogical switch being closed after this time is an indication of its failure. The additional 0.3 seconds provides a hysteresis that prevents transient alarms.

Traceability: Developed from SRATS-11. Developed into LLR-51, LLR-52, LLR-53, LLR-54 and LLR-55.

HLR-12 *Hydraulic Circuit Unpressurized Failure Detection*. 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.

Rationale: The hydraulic circuit should be pressurized in less than 2 seconds after the general electro-valve has been stimulated. The hydraulic circuit being unpressurized after this time is an indication of its failure. This avoids CFC-3.

Traceability: Developed from SRATS-12. Developed into LLR-44, LLR-56 and LLR-57.

HLR-13 *Hydraulic Circuit Pressurized Failure Detection*. Once 10 seconds have elapsed since the General EV Actuation Command controllable variable was set to Close and the overall value of the Hydraulic Circuit Pressure monitorable variables is still greater than or equal to 30,000 kPa and less than 35,000 kPa, the LGCS shall detect a failure of the general hydraulic electro-valve.

Rationale: The hydraulic circuit should be unpressurized in less than 10 seconds after the general electro-valve has stopped being stimulated. The hydraulic circuit being pressurized after this time is an indication of its failure. This avoids CFC-3.

Traceability: Developed from SRATS-13. Developed into LLR-58 and LLR-59.

HLR-14 *Door Opening Failure Detection*. Once 7 seconds have elapsed since the Door Opening EV Actuation Command controllable variable was set to Open and the overall value of the Door Closed Status monitorable variables for any wheel assembly is still Closed or the overall value of the Door Open Status monitorable variables for any wheel assembly is still Not Open, the LGCS shall detect a failure of the doors.

Rationale: The doors should be locked open in less than 7 seconds after stimulating the door opening electro-valve. The doors being closed after this time is an indication of their failure. This avoids CFC-3.

Traceability: Developed from SRATS-14. Developed into LLR-60, LLR-61 and LLR-62.

HLR-15 *Door Closing Failure Detection*. Once 7 seconds have elapsed since the `Door Closing EV Actuation Command` controllable variable was set to `Open` and the overall value of the `Door Open Status` monitorable variables for any wheel assembly is still `Open` or the overall value of the `Door Closed Status` monitorable variable for any wheel assembly is still `Not Closed`, the LGCS shall detect a failure of the doors.

Rationale: The doors should be locked closed in less than 7 seconds after stimulating the door closing electro-valve. The doors being open after this time is an indication of their failure. This avoids CFC-3.

Traceability: Developed from SRATS-15. Developed into LLR-63, LLR-64 and LLR-65.

HLR-16 *Gear Retraction Failure Detection*. Once 10 seconds have elapsed since the `Gear Retraction EV Actuation Command` controllable variable was set to `Open` and the overall value of the `Gear Extended Status` monitorable variable for any wheel assembly is still `Extended` or the overall value of the `Gear Retracted Status` monitorable variable for any wheel assembly is still `Not Retracted`, the LGCS shall detect a failure of the gears.

Rationale: The gears should be retracted in less than 10 seconds after stimulating the gear retraction electro-valve. The gears being extended after this time is an indication of their failure. This avoids CFC-3.

Traceability: Developed from SRATS-16. Developed into LLR-66, LLR-67 and LLR-68.

HLR-17 *Gear Extension Failure Detection*. Once 10 seconds have elapsed since the Gear Extension EV Actuation Command controllable variable was set to Open and the overall value of the Gear Retracted Status monitorable variable for any wheel assembly is still Retracted or the overall value of the Gear Extended Status monitorable variable for any wheel assembly is still Not Extended, the LGCS shall detect a failure of the gears.

Rationale: The gears should be extended in less than 10 seconds after stimulating the gear extension electro-valve. The gears being retracted after this time is an indication of their failure. This avoids CFC-3.

Traceability: Developed from SRATS-17. Developed into LLR-69, LLR-70 and LLR-71.

HLR-18 *Failure Detected*. When the LGCS is currently executing a retraction or extension sequence and a failure is detected, the LGCS shall halt the currently executing sequence and set the Feedback controllable variable to Red.

Rationale: If a failure is detected as specified in HLR-1, HLR-5, HLR-11, HLR-12, HLR-13, HLR-14, HLR-15, HLR-16 and HLR-17, the system should not be not be trusted to perform correctly and, therefore, the LGCS shall not be functional to avoid CFC-2 and CFC-3.

Traceability: Developed from SRATS-18. Developed into LLR-72, LLR-73, LLR-74, LLR-75 and LLR-76.

LLR-1 If the SensorManager receives data from a sensor, the validateSensorData function shall be activated.

Traceability: Developed from HLR-1.

Apportioned to: SensorManager.

LLR-2 If the validateSensorData function is active, it shall process the three readings associated to the sensor data received.

Traceability: Developed from HLR-1.

Apportioned to: SensorManager.

LLR-3 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.

Traceability: Developed from HLR-1.

Apportioned to: SensorManager.

LLR-4 If the validateSensorData function is active, the three readings are valid and do not have equal values, the overall sensor value shall be any of the readings' value and be invalid.

Traceability: Developed from HLR-1.

Apportioned to: SensorManager.

LLR-5 If the validateSensorData function is active, the three readings are valid and only two of them have equal values, the reading with the different value shall be considered invalid and not considered in the future. The overall sensor value shall be the common value of the two remaining valid readings and be valid.

Traceability: Developed from HLR-1.

Apportioned to: SensorManager.

LLR-6 If the validateSensorData function is active, one reading was eliminated, and the two remaining readings are valid and do not have equal values, the overall sensor value shall be any of the readings' value and be invalid.

Traceability: Developed from HLR-1.

Apportioned to: SensorManager.

LLR-7 If the validateSensorData function is active, one reading is invalid, and the two remaining readings are valid and have equal values, the invalid reading shall not be

considered in the future. The overall sensor value shall be the common value of the two remaining valid readings and be valid.

Traceability: Developed from HLR-1.

Apportioned to: SensorManager.

LLR-8 If the validateSensorData function is active and at least two readings are invalid, the overall sensor value shall be any of the readings' value and be invalid.

Traceability: Developed from HLR-1.

Apportioned to: SensorManager.

LLR-9 If the overall sensor value is invalid, the FailureEvent shall be raised.

Traceability: Developed from HLR-1.

Apportioned to: SensorManager.

LLR-10 If the overall sensor value is valid, it shall be made available to the SequenceController and the SensorManager shall wait to receive data from a sensor.

Traceability: Developed from HLR-1.

Apportioned to: SensorManager.

LLR-11 If the retractGears or the extendGears function becomes active, the GearsInTransitEvent shall be raised.

Traceability: Developed from HLR-2 and HLR-3.

Apportioned to: SequenceController.

LLR-12 If the GearsInTransitEvent is raised, the Feedback controllable variable shall be set to Amber.

Traceability: Developed from HLR-2 and HLR-3.

Apportioned to: PilotInterfaceManager.

LLR-13 If the moveGears function is active, the General EV Actuation Command controllable variable shall be set to Open.

Traceability: Developed from HLR-2 and HLR-3.

Apportioned to: SequenceController.

LLR-14 If the PressurizationEvent and the OpenGeneralEVDelayTimeoutEvent are raised, the Door Opening EV Actuation Command controllable variable shall be set to Open and the waitForDoorsOpen function shall be activated.

Traceability: Developed from HLR-2, HLR-3, HLR-6 and HLR-7.

Apportioned to: SequenceController.

LLR-15 If the waitForDoorsOpen function is active and the overall value of the Door Open Status monitorable variables for any wheel assembly is Not Open or the overall value of the Door Closed Status monitorable variable for any wheel assembly is Closed, the waitForDoorsOpen function shall remain active until the DoorsOpeningTimeoutEvent is raised.

Traceability: Developed from HLR-2 and HLR-3.

Apportioned to: SequenceController.

LLR-16 If the waitForDoorsOpen function is active and the overall value of the Door Open Status monitorable variables for all wheel assemblies is Open and the overall value of the Door Closed Status monitorable variables for all wheel assemblies is Not Closed, the waitForDoorsOpen function shall end and the Door Opening EV Actuation Command controllable variable set to Closed.

Traceability: Developed from HLR-2 and HLR-3.

Apportioned to: SequenceController.

LLR-17 If the waitForDoorsOpen function ended, the Desired Gear Position is equal to Up, the overall value of the Gear Shock Absorber Status monitorable variables

for all wheel assemblies is Relaxed and the ConsecutiveSpecificEVDelayTimeoutEvent is raised, the Gear Retraction EV Actuation Command controllable variable shall be set to Open and the waitForGearsRetracted function shall be activated.

Traceability: Developed from HLR-2 and HLR-8.

Apportioned to: SequenceController.

LLR-18 If the overall value of the Gear Shock Absorber Status monitorable variables for any wheel assembly is Not Relaxed, the PlaneOnGroundEvent shall be raised.

Traceability: Developed from HLR-2.

Apportioned to: SequenceController.

LLR-19 If the waitForGearsRetracted function is active and the overall value of the Gear Retracted Status monitorable variables for any wheel assembly is Not Retracted or the overall value of the Gear Extended Status monitorable variables for any wheel assembly is Extended, the waitForGearsRetracted function shall remain active until the GearsRetractionTimeoutEvent is raised.

Traceability: Developed from HLR-2.

Apportioned to: SequenceController.

LLR-20 If the waitForGearsRetracted function is active and the overall value of the Gear Retracted Status monitorable variables for all wheel assemblies is Retracted and the overall value for the Gear Extended Status monitorable variables for all wheel assemblies is Not Extended, the waitForGearsRetracted function shall end and the Gear Retraction EV Actuation Command controllable variable shall be set to Closed.

Traceability: Developed from HLR-2.

Apportioned to: SequenceController.

LLR-21 If the waitForGearsRetracted function ended, the GearsRetractedEvent shall be raised.

Traceability: Developed from HLR-2 and HLR-4.

Apportioned to: SequenceController.

LLR-22 If the GearsRetractedEvent is raised, the Feedback controllable variable shall be set to Off.

Traceability: Developed from HLR-2.

Apportioned to: PilotInterfaceManager.

LLR-23 If the PlaneOnGroundEvent is raised, the GearsExtendedEvent shall be raised.

Traceability: Developed from HLR-2.

Apportioned to: SequenceController.

LLR-24 If the waitForGearsRetracted function ended or the PlaneOnGroundEvent is raised, and the ConsecutiveSpecificEVDelayTimeoutEvent is raised, the Door Closing EV Actuation Command controllable variable shall be set to Open and the waitForDoorsClosed function shall be activated.

Traceability: Developed from HLR-2 and HLR-8.

Apportioned to: SequenceController.

LLR-25 If the waitForDoorsClosed function is active and the overall value of the Door Closed Status monitorable variables for any wheel assembly is Not Closed or the overall value of the Door Open Status monitorable variables for any wheel assembly is Open, the waitForDoorsClosed function shall remain active until the DoorsClosingTimeoutEvent is raised.

Traceability: Developed from HLR-2 and HLR-3.

Apportioned to: SequenceController.

LLR-26 If the waitForDoorsClosed function is active and the overall value of the Door Closed Status monitorable variables for all wheel assemblies is Closed and the overall value of the Door Open Status monitorable variables for all wheel assemblies is Not Open, the waitForDoorsClosed function shall end and the Door Closing EV Actuation Command controllable variable shall be set to Closed.

Traceability: Developed from HLR-2 and HLR-3.

Apportioned to: SequenceController.

LLR-27 If the waitForDoorsClosed function ended and the CloseGeneralEVDelayTimeoutEvent is raised, the General EV Actuation Command controllable variable shall be set to Closed.

Traceability: Developed from HLR-2, HLR-3 and HLR-9.

Apportioned to: SequenceController.

LLR-28 If the waitForDoorsOpen function ended, the Desired Gear Position is equal to Down and the ConsecutiveSpecificEVDelayTimeoutEvent is raised, the Gear Extension EV Actuation Command controllable variable shall be set to Open and the waitForGearsExtended function shall be activated.

Traceability: Developed from HLR-3 and HLR-8.

Apportioned to: SequenceController.

LLR-29 If the waitForGearsExtended function is active and the overall value of the Gear Extended Status monitorable variables for any wheel assembly is Not Extended or the overall value of the Gear Retracted Status monitorable variables for any wheel assembly is Retracted, the waitForGearsExtended function shall remain active until the GearsExtensionTimeoutEvent is raised.

Traceability: Developed from HLR-3.

Apportioned to: SequenceController.

LLR-30 If the waitForGearsExtended function is active and the overall value of the Gear Extended Status monitorable variables for all wheel assemblies is Extended and the overall value of the Gear Retracted Status monitorable variables for all wheel assemblies is Not Retracted, the waitForGearsExtended function shall end and the Gear Extension EV Actuation Command controllable variable shall be set to Closed.

Traceability: Developed from HLR-3.

Apportioned to: SequenceController.

LLR-31 If the waitForGearsExtended function ended, the GearsExtendedEvent shall be raised.

Traceability: Developed from HLR-3.

Apportioned to: SequenceController.

LLR-32 If the GearsExtendedEvent is raised, the Feedback controllable variable shall be set to Green.

Traceability: Developed from HLR-3 and HLR-4.

Apportioned to: PilotInterfaceManager.

LLR-33 If the waitForGearsExtended function ended and the ConsecutiveSpecificEVDelayTimeoutEvent is raised, the Door Closing EV Actuation Command controllable variable shall be set to Open and the waitForDoorsClosed function shall be activated.

Traceability: Developed from HLR-3 and HLR-8.

Apportioned to: SequenceController.

LLR-34 If the LGCS is currently executing a sequence and a Desired Gear Position is received, the RevertEvent shall be raised.

Traceability: Developed from HLR-4.

Apportioned to: PilotInterfaceManager.

LLR-35 If the RevertEvent is raised, all the actions that were executed shall be reverted.

Traceability: Developed from HLR-4.

Apportioned to: SequenceController.

LLR-36 If the retraction sequence is reverted, the GearsExtendedEvent shall be raised.

Traceability: Developed from HLR-4.

Apportioned to: SequenceController.

LLR-37 If the extension sequence is reverted, the GearsRetractedEvent shall be raised.

Traceability: Developed from HLR-4.

Apportioned to: SequenceController.

LLR-38 If a Desired Gear Position is received, the waitForAnalogicalSwitchClosed function shall be activated.

Traceability: Developed from HLR-5.

Apportioned to: SequenceController.

LLR-39 If the waitForAnalogicalSwitchClosed function is active and the overall value of the Analogical Switch Status monitorable variables is Open, the waitForAnalogicalSwitchClosed function shall remain active until the AnalogicalSwitchClosingTimeoutEvent is raised.

Traceability: Developed from HLR-5.

Apportioned to: SequenceController.

LLR-40 If the waitForAnalogicalSwitchClosed function is active and the overall value of the Analogical Switch Status monitorable variables is Closed, the waitForAnalogicalSwitchClosed function shall end and the moveGears function shall be activated.

Traceability: Developed from HLR-2, HLR-3 and HLR-5.

Apportioned to: SequenceController.

LLR-41 If 1 second has elapsed since a Desired Gear Position was received, the AnalogicalSwitchClosingTimeoutEvent shall be raised.

Traceability: Developed from HLR-5.

Apportioned to: SequenceController.

LLR-42 If the AnalogicalSwitchClosingTimeoutEvent is raised and the overall value of the Analogical Switch Status monitorable variables is Open, the FailureEvent shall be raised.

Traceability: Developed from HLR-5.

Apportioned to: SequenceController.

LLR-43 If the General EV Actuation Command controllable variable is set to Open, the waitForHydraulicPressure function shall be activated.

Traceability: Developed from HLR-6.

Apportioned to: SequenceController.

LLR-44 If the waitForHydraulicPressure function is active and the overall value of the Hydraulic Circuit Pressure monitorable variable is less than 30,000 kPa, the waitForHydraulicPressure function shall remain active until the PressurizationTimeoutEvent is raised.

Traceability: Developed from HLR-6 and HLR-12.

Apportioned to: SequenceController.

LLR-45 If the waitForHydraulicPressure function is active and the overall value of the Hydraulic Circuit Pressure monitorable variables is greater than or equal to 30,000 kPa and less than 35,000 kPa, the waitForHydraulicPressure function shall end and the PressurizationEvent shall be raised.

Traceability: Developed from HLR-6.

Apportioned to: SequenceController.

LLR-46 If 0.2 seconds have elapsed since the General EV Actuation Command controllable variable was set to Open, the OpenGeneralEVDelayTimeoutEvent shall be raised.

Traceability: Developed from HLR-7.

Apportioned to: SequenceController.

LLR-47 If 0.2 seconds have elapsed since a specific EV Actuation Command controllable variable was set to Closed, the ConsecutiveSpecificEVDelayTimeoutEvent shall be raised.

Traceability: Developed from HLR-8.

Apportioned to: SequenceController.

LLR-48 If 1 second has elapsed since a specific EV Actuation Command controllable variable was set to Closed, the CloseGeneralEVDelayTimeoutEvent shall be raised.

Traceability: Developed from HLR-9.

Apportioned to: SequenceController.

LLR-49 If 0.1 seconds have elapsed since a specific EV Actuation Command controllable variable was set to Closed, the ContrarySpecificEVDelayTimeoutEvent shall be raised.

Traceability: Developed from HLR-10.

Apportioned to: SequenceController.

LLR-50 If the ContrarySpecificEVDelayTimeoutEvent is raised, the contrary specific EV Actuation command controllable variable can be set to Open.

Traceability: Developed from HLR-10.

Apportioned to: SequenceController.

LLR-51 If 28 seconds have elapsed since the moveGears function is active, the waitForAnalogicalSwitchOpen function shall be activated.

Traceability: Developed from HLR-11.

Apportioned to: SequenceController.

LLR-52 If the waitForAnalogicalSwitchOpen function is active and the overall value of the Analogical Switch Status monitorable variables is Closed, the waitForAnalogicalSwitchOpen function shall remain active until the AnalogicalSwitchOpeningTimeoutEvent is raised.

Traceability: Developed from HLR-11.

Apportioned to: SequenceController.

LLR-53 If the waitForAnalogicalSwitchOpen function is active and the overall value of the Analogical Switch Status monitorable variables is Open, the waitForAnalogicalSwitchOpen function shall end and the LGCS shall wait to receive a Desired Gear Position.

Traceability: Developed from HLR-11.

Apportioned to: SequenceController.

LLR-54 If 1.5 seconds have elapsed after the waitForAnalogicalSwitchOpen function became active, the AnalogicalSwitchOpenTimeoutEvent shall be raised.

Traceability: Developed from HLR-11.

Apportioned to: SequenceController.

LLR-55 If the AnalogicalSwitchOpenTimeoutEvent is raised and the overall value of the Analogical Switch Status monitorable variables is Closed, the FailureEvent shall be raised.

Traceability: Developed from HLR-11.

Apportioned to: SequenceController.

LLR-56 If 2 seconds have elapsed since the General EV controllable variable was set to Open, the PressurizationTimeoutEvent shall be raised.

Traceability: Developed from HLR-12.

Apportioned to: SequenceController.

LLR-57 If the PressurizationTimeoutEvent is raised and the overall value of the Hydraulic Circuit Pressure monitorable variables is less than 30,000 kPa, the FailureEvent shall be raised.

Traceability: Developed from HLR-12.

Apportioned to: SequenceController.

LLR-58 If 10 seconds have elapsed since the General EV controllable variable was set to Close, the DepressurizationTimeoutEvent shall be raised.

Traceability: Developed from HLR-13.

Apportioned to: SequenceController.

LLR-59 If the DepressurizationTimeoutEvent is raised and the overall value of the Hydraulic Circuit Pressure monitorable variables is greater than or equal to 30,000 kPa and less than 35,000 kPa, the FailureEvent shall be raised.

Traceability: Developed from HLR-13.

Apportioned to: SequenceController.

LLR-60 If 7 seconds have elapsed since the Door Opening EV Actuation Command controllable variable was set to Open, the DoorsOpeningTimeoutEvent shall be raised.

Traceability: Developed from HLR-14.

Apportioned to: SequenceController.

LLR-61 If the DoorsOpeningTimeoutEvent is raised and the overall value of the Door Closed Status monitorable variables for any wheel assembly is Closed, the FailureEvent shall be raised.

Traceability: Developed from HLR-14.

Apportioned to: SequenceController.

LLR-62 If the DoorsOpeningTimeoutEvent is raised and the overall value of the Door Open Status monitorable variables for any wheel assembly is Not Open, the FailureEvent shall be raised.

Traceability: Developed from HLR-14.

Apportioned to: SequenceController.

LLR-63 If 7 seconds have elapsed since the Door Closing EV Actuation Command was set to Open, the DoorsClosingTimeoutEvent shall be raised.

Traceability: Developed from HLR-15.

Apportioned to: SequenceController.

LLR-64 If the DoorsClosingTimeoutEvent is raised and the overall value of the Door Open Status monitorable variables for any wheel assembly is Open, the FailureEvent shall be raised.

Traceability: Developed from HLR-15.

Apportioned to: SequenceController.

LLR-65 If the DoorsClosingTimeoutEvent is raised and the overall value of the Door Closed Status monitorable variables for any wheel assembly is Not Closed, the FailureEvent shall be raised.

Traceability: Developed from HLR-15.

Apportioned to: SequenceController.

LLR-66 If 10 seconds have elapsed since the Gear Retraction EV Actuation Command controllable variable was set to Open, the GearsRetractionTimeoutEvent shall be raised.

Traceability: Developed from HLR-16.

Apportioned to: SequenceController.

LLR-67 If the GearsRetractionTimeoutEvent is raised and the overall value of the Gear Extended Status monitorable variables for any wheel assembly is Extended, the FailureEvent shall be raised.

Traceability: Developed from HLR-16.

Apportioned to: SequenceController.

LLR-68 If the GearsRetractionTimeoutEvent is raised and the overall value of the Gear Retracted Status monitorable variables for any wheel assembly is Not Retracted, the FailureEvent shall be raised.

Traceability: Developed from HLR-16.

Apportioned to: SequenceController.

LLR-69 If 10 seconds have elapsed since the Gear Extension EV Actuation Command controllable variable was set to Open, the GearsExtensionTimeoutEvent shall be raised.

Traceability: Developed from HLR-17.

Apportioned to: SequenceController.

LLR-70 If the GearsExtensionTimeoutEvent is raised and the overall value of the Gear Retracted Status monitorable variables for any wheel assembly is Retracted, the FailureEvent shall be raised.

Traceability: Developed from HLR-17.

Apportioned to: SequenceController.

LLR-71 If the GearsExtensionTimeoutEvent is raised and the overall value of the Gear Extended Status monitorable variables for any wheel assembly is Not Extended, the FailureEvent shall be raised.

Traceability: Developed from HLR-17.

Apportioned to: SequenceController.

LLR-72 If the FailureEvent is raised and the LGCS is currently executing a retraction or extension sequence, the HaltActiveSequenceEvent shall be raised.

Traceability: Developed from HLR-18.

Apportioned to: SequenceController.

LLR-73 If the FailureEvent is raised, the FailureDetectedEvent shall be raised.

Traceability: Developed from HLR-18.

Apportioned to: OperatingModeManager.

LLR-74 If the FailureDetectedEvent is raised, the Feedback controllable variable shall be set to Red.

Traceability: Developed from HLR-18.

Apportioned to: PilotInterfaceManager.

LLR-75 If the HaltActiveSequenceEvent is raised, the haltActiveSequence procedure shall be activated.

Traceability: Developed from HLR-18.

Apportioned to: SequenceController.

LLR-76 If the haltActiveSequence procedure is active, the current sequence of actions shall be halted.

Traceability: Developed from HLR-18.

Apportioned to: SequenceController.

APPENDIX III

MAPPING RULES AND DESIGN GUIDELINES OF *CHECSDM4USS*

Table-A III-1 Mapping rule mr_us_01 for UML class and Simulink subsystem.

Mapping Rule (ID: Name)	mr_us_01: UML class and Simulink subsystem
When	The UML class and the Simulink subsystem have similar names.
Where	
(1)	Input parameters of operations in the UML classifier's provided interface and inputs of the Simulink subsystem block are matched (referenced rule: mr_us_05, see Table III-5).
(2)	Output parameters of operations in the UML classifier's required interface and inputs of the Simulink subsystem block are matched (referenced rule: mr_us_06, see Table III-6).
(3)	Input parameters of operations in the UML classifier's required interface and outputs of the Simulink subsystem block are matched (referenced rule: mr_us_07, see Table III-7).
(4)	Output parameters of operations in the UML classifier's provided interface and outputs of the Simulink subsystem block are matched (referenced rule: mr_us_08, see Table III-8).
(5)	Nested classifiers and nested subsystem blocks are matched (referenced rule: mr_us_01, see Table III-1).

Table-A III-2 Mapping rule mr_us_02 for UML class and Stateflow chart.

Mapping Rule (ID: Name)	mr_us_02: UML class and Stateflow chart
When	The UML class and the Stateflow chart have similar names.
Where	
(1)	Input parameters of operations in the UML classifier's provided interface and inputs of the Stateflow chart are matched (referenced rule: mr_us_05, see Table III-5).
(2)	Output parameters of operations in the UML classifier's required interface and inputs of the Stateflow chart are matched (referenced rule: mr_us_06, see Table III-6).
(3)	Input parameters of operations in the UML classifier's required interface and outputs of the Stateflow chart are matched (referenced rule: mr_us_07, see Table III-7).
(4)	Output parameters of operations in the UML classifier's provided interface and outputs of the Stateflow chart are matched (referenced rule: mr_us_08, see Table III-8).

Table-A III-3 Mapping rule mr_us_03 for UML component and Simulink subsystem.

Mapping Rule (ID: Name)	mr_us_03: UML component and Simulink subsystem
When	The UML component and the Simulink subsystem have similar names.
Where	
(1)	Input parameters of operations in the UML classifier's provided interface and inputs of the Simulink subsystem block are matched (referenced rule: mr_us_05, see Table III-5).
(2)	Output parameters of operations in the UML classifier's required interface and inputs of the Simulink subsystem block are matched (referenced rule: mr_us_06, see Table III-6).
(3)	Input parameters of operations in the UML classifier's required interface and outputs of the Simulink subsystem block are matched (referenced rule: mr_us_07, see Table III-7).
(4)	Output parameters of operations in the UML classifier's provided interface and outputs of the Simulink subsystem block are matched (referenced rule: mr_us_08, see Table III-8).
(5)	Nested classifiers and nested subsystem blocks are matched (referenced rule: mr_us_03, see Table III-3).

Table-A III-4 Mapping rule mr_us_04 for UML component and Stateflow chart.

Mapping Rule (ID: Name)	mr_us_04: UML component and Stateflow chart
When	The UML component and the Stateflow chart have similar names.
Where	
(1)	Input parameters of operations in the UML classifier's provided interface and inputs of the Stateflow chart are matched (referenced rule: mr_us_05, see Table III-5).
(2)	Output parameters of operations in the UML classifier's required interface and inputs of the Stateflow chart are matched (referenced rule: mr_us_06, see Table III-6).
(3)	Input parameters of operations in the UML classifier's required interface and outputs of the Stateflow chart are matched (referenced rule: mr_us_07, see Table III-7).
(4)	Output parameters of operations in the UML classifier's provided interface and outputs of the Stateflow chart are matched (referenced rule: mr_us_08, see Table III-8).

Table-A III-5 Mapping rule mr_us_05 for UML input parameter and Simulink block input.

Mapping Rule (ID: Name)	mr_us_05: UML input parameter and Simulink block input
When	
(1)	The input parameter's name is similar to the block input's name.
(2)	The input parameter's data type is similar to the block input's data type.

Note: Empty compartments are omitted to keep the table uncluttered.

Table-A III-6 Mapping rule mr_us_06 for UML output parameter and Simulink block input.

Mapping Rule (ID: Name)	mr_us_06: UML output parameter and Simulink block input
When	
(1)	The output parameter's name is similar to the block input's name.
(2)	The output parameter's data type is similar to the block input's data type.

Note: Empty compartments are omitted to keep the table uncluttered.

Table-A III-7 Mapping rule mr_us_07 for UML input parameter and Simulink block output.

Mapping Rule (ID: Name)	mr_us_07: UML input parameter and Simulink block output
When	
(1)	The input parameter's name is similar to the block output's name.
(2)	The input parameter's data type is similar to the block output's data type.

Note: Empty compartments are omitted to keep the table uncluttered.

Table-A III-8 Mapping rule mr_us_08 for UML output parameter and Simulink block output.

Mapping Rule (ID: Name)	mr_us_08: UML output parameter and Simulink block output
When	
(1)	The output parameter's name is similar to the block output's name.
(2)	The output parameter's data type is similar to the block output's data type.

Note: Empty compartments are omitted to keep the table uncluttered.

Table-A III-9 Mapping rule mr_us_09 for UML state machine and Stateflow chart.

Mapping Rule (ID: Name)	mr_us_09: UML state machine and Stateflow chart
When	The UML state machine and Stateflow chart have similar names.
Where	
(1)	Regions in the UML state machine and parallel states at the topmost level of the Stateflow chart are matched (referenced rule: mr_us_11, see Table III-11). This clause is dropped when the UML state machine has only one region.
(2)	States in the UML state machine and states in the Stateflow chart are matched (referenced rule: mr_us_12, see Table III-12).
(3)	Choice pseudostates in the UML state machine and connective junctions in the Stateflow chart are matched (referenced rule: mr_us_13, see Table III-13).
(4)	Fork pseudostates in the UML state machine and Stateflow composite states in the Stateflow chart are matched (referenced rule: mr_us_14, see Table III-14).
(5)	Join pseudostates in the UML state machine and Stateflow composite states in the Stateflow chart are matched (referenced rule: mr_us_15, see Table III-15).
(6)	Initial states in the UML state machine and default transitions in the Stateflow chart are matched (referenced rule: mr_us_16, see Table III-16).
(7)	Transitions in the UML state machine and transitions in the Stateflow chart are matched (referenced rule: mr_us_17, see Table III-17).

Table-A III-10 Mapping rule mr_us_10 for UML composite state and Stateflow composite state.

Mapping Rule (ID: Name)	mr_us_10: UML composite state and Stateflow composite state
When	The UML composite state and Stateflow composite state have similar names.
Where	
(1)	Regions in the UML composite state and parallel states at the topmost level of the Stateflow composite state are matched (referenced rule: mr_us_11, see Table III-11). This clause is dropped when the UML composite state has only one region.
(2)	States within the UML composite state and states within the Stateflow composite state are matched (referenced rule: mr_us_12, see Table III-12).
(3)	Choice pseudostates within the UML composite state and connective junctions within the Stateflow composite state are matched (referenced rule: mr_us_13, see Table III-13).
(4)	Fork pseudostates within the UML composite state and Stateflow composite states within the Stateflow composite state are matched (referenced rule: mr_us_14, see Table III-14).
(5)	Join pseudostates within the UML composite state and Stateflow composite states within the Stateflow composite state are matched (referenced rule: mr_us_15, see Table III-15).
(6)	Initial states within the UML composite state and default transitions within the Stateflow composite state are matched (referenced rule: mr_us_16, see Table III-16).
(7)	Transitions within the UML composite state and transitions within the Stateflow composite state are matched (referenced rule: mr_us_17, see Table III-17).

Table-A III-11 Mapping rule mr_us_11 for UML region and Stateflow parallel state.

Mapping Rule (ID: Name)	mr_us_11: UML region and Stateflow parallel state
When	
(1)	The containing Stateflow chart or composite state has parallel (AND) decomposition.
(2)	The Stateflow parallel state is at the topmost level of its containing chart/composite state.
(3)	the UML region and Stateflow parallel state have similar names.
Where	
(1)	Nested elements (*i.e.* states, transitions, pseudostates) are matched (referenced rules: mr_us_10 through mr_us_17, see Tables III-10 through III-17).

Table-A III-12 Mapping rule mr_us_12 for UML state and Stateflow state.

Mapping Rule (ID: Name)	mr_us_12: UML state and Stateflow state
When	The UML state and Stateflow state have similar names.
Where	
(1)	Incoming and outgoing transitions are matched (referenced rule: mr_us_17, see Table III-17).
(2)	Entry, do and exit actions are matched.

Table-A III-13 Mapping rule mr_us_13 for UML choice pseudostate and Stateflow junction.

Mapping Rule (ID: Name)	mr_us_13: UML choice pseudostate and Stateflow junction
Where	
(1)	Incoming and outgoing transitions are matched (referenced rule: mr_us_17, see Table III-17).

Note: Empty compartments are omitted to keep the table uncluttered.

Table-A III-14 Mapping rule mr_us_14 for UML fork pseudostate and Stateflow composite state.

Mapping Rule (ID: Name)	mr_us_14: UML fork pseudostate and Stateflow composite state
When	
(1)	The Stateflow composite state has parallel (AND) decomposition.
(2)	The UML fork pseudostates and Stateflow composite state have similar names (see guideline av_us_17, Table III-37).
Where	
(1)	Incoming transitions are matched (referenced rule: mr_us_17, see Table III-17).

Table-A III-15 Mapping rule mr_us_15 for UML join pseudostate and Stateflow composite state.

Mapping Rule (ID: Name)	mr_us_15: UML join pseudostate and Stateflow composite state
When	
(1)	The Stateflow composite state has parallel (AND) decomposition.
(2)	The UML join pseudostates and Stateflow composite state have similar names (see guideline av_us_17, Table III-37).
Where	
(1)	Outgoing transitions are matched (referenced rule: mr_us_17, see Table III-17).

Table-A III-16 Mapping rule mr_us_16 for UML default transition and Stateflow default transition.

Mapping Rule (ID: Name)	mr_us_16: UML default transition and Stateflow default transition
When	The target state of the outgoing transition from the UML initial state and the target state of the Stateflow default transition are matched.

Note: Empty compartments are omitted to keep the table uncluttered.

Table-A III-17 Mapping rule mr_us_17 for UML transition and Stateflow transition.

Mapping Rule (ID: Name)	mr_us_17: UML transition and Stateflow transition
When	
(1)	The target vertexes have similar names.
(2)	the source vertexes have similar names.
Where	
(1)	Triggers, guards and actions are matched (referenced rules: mr_us_18 through mr_us_20, see Tables III-18 through III-20).

Table-A III-18 Mapping rule mr_us_18 for UML transition trigger and Stateflow transition trigger.

Mapping Rule (ID: Name)	mr_us_18: UML transition trigger and Stateflow transition trigger
When	The trigger expressions are matched (see guideline av_us_10, Table III-30).

Note: Empty compartments are omitted to keep the table uncluttered.

Table-A III-19 Mapping rule mr_us_19 for UML transition guard and Stateflow transition guard.

Mapping Rule (ID: Name)	mr_us_19: UML transition guard and Stateflow transition guard
When	The guard expressions are matched (see guidelines av_us_11 and av_us_12, Tables III-31 and III-32).

Note: Empty compartments are omitted to keep the table uncluttered.

Table-A III-20 Mapping rule mr_us_20 for UML transition actions and Stateflow transition actions.

Mapping Rule (ID: Name)	mr_us_20: UML transition actions and Stateflow transition actions
When	The actions' expressions are matched (see guideline av_us_15, Table III-35).

Note: Empty compartments are omitted to keep the table uncluttered.

Table-A III-21 Guideline av_us_01: Mixed use of UML, Simulink and Stateflow.

Guideline (ID: Title)	av_us_01: Mixed use of UML, Simulink and Stateflow
Priority	Recommended
Scope	UML, Simulink and Stateflow
Prerequisites	None
Description	
	The choice of using UML, Simulink or Stateflow, or a mix of them to model all the system or given portions of it should be driven by the nature of the system being modeled and the purposes of the models.
•	If the system (or the portions of it) primarily involves software, use UML components and classes to characterize the structure and behaviour of the software.
•	If the system (or the portions of it) involves both software and hardware elements, use UML components and classes to characterize the structure and behaviour of the software, and use Simulink and Stateflow to characterize the structure and behaviour of the hardware elements.
•	If the behaviour of the system (or the portions of it) primarily involves modal logic with a combination of past and present logical conditions, use UML state machines, Stateflow charts or a mix of them.
•	If a behaviour segment primarily involves behaviour that may execute concurrently, use UML state machines.
•	If the behaviour of the system (or the portions of it) primarily involves if-then-else statements, use Stateflow truth table charts.
Rationale	
	Embedded control software are highly heterogeneous, possessing very different characteristics. In order to cope with their complexities, these characteristics should be described using diverse modelling mechanisms, like UML, Simulink and Stateflow. This guideline is intended to manage the resulting design heterogeneity in order to facilitate understanding and communication without hindering verification and certification.
See Also	
•	Guideline av_us_02: Definition of a naming convention (see Table III-22)
•	MAAB Guideline na_0006: Guidelines for mixed use of Simulink and Stateflow
•	MAAB Guideline na_0007: Guidelines for use of Flow Charts, Truth Tables and State Machines

Table-A III-22 Guideline av_us_02: Definition of a naming convention.

Guideline (ID: Title)	av_us_02: Definition of a naming convention
Priority	Mandatory
Scope	UML, Simulink and Stateflow
Prerequisites	
•	MAAB Guideline ar_0001: Filenames
•	MAAB Guideline ar_0002: Directory names
•	MAAB Guideline na_0035: Adoption of naming conventions
•	MAAB Guideline jc_0201: Usable characters for Subsystem names
•	MAAB Guideline jc_0211: Usable characters for Inport block and Outport block
•	MAAB Guideline jc_0231: Usable characters for block names
•	MAAB Guideline na_0014: Use of local language in Simulink and Stateflow
•	UML 2.5.1 subclause 9.2.4 Notation (Classifiers)
•	UML 2.5.1 subclause 9.5.4 Notation (Properties)
•	UML 2.5.1 subclause 9.6.4 Notation (Operations)
•	UML 2.5.1 subclause 10.4.4 Notation (Interfaces)
•	UML 2.5.1 subclause 11.4.4 Notation (Classes)
•	UML 2.5.1 subclause 11.6.4 Notation (Components)
•	UML 2.5.1 subclause 14.2.4 Notation (State Machines)
Description	
	It is not possible to define a single naming convention applicable to the internal processes of all organizations. However, within an organization it is mandatory to follow a single, consistent naming convention that is compatible across UML, Simulink and Stateflow models. Such naming convention must provide guidance for naming Simulink subsystem blocks, Simulink basic blocks, Simulink input and output ports, Stateflow charts, Stateflow states, Stateflow data (input, output and local), and UML`NamedElement`s.
Rationale	
	This guideline is intended to make mandatory a naming convention across UML, Simulink and Stateflow models in order to reduce ambiguities when mapping UML constructs and Simulink or Stateflow constructs.

Table-A III-23 Guideline av_us_03: Naming of elements in UML models.

Guideline (ID: Title)	av_us_03: Naming of elements in UML models
Priority	Mandatory
Scope	UML
Prerequisites	None
Description	
	All `NamedElement`s in a UML model must have a name.
Rationale	
	Finding matches between UML, Simulink and Stateflow models relies on naming.

Table-A III-24 Guideline av_us_04: Naming of UML fork and join pseudostates.

Guideline (ID: Title)	av_us_04: Naming of UML fork and join pseudostates
Priority	Mandatory
Scope	UML
Prerequisites	None
Description	
	The names of UML fork and join pseudostates must be the same.
Rationale	
	Sharing the same name root facilitates their matching to an equivalent Stateflow construct.
See Also	
•	Guideline av_us_14: Fork and join behaviour in Stateflow (see Table III-41)

Table-A III-25 Guideline av_us_05: Naming of elements in Simulink / Stateflow models.

Guideline (ID: Title)	av_us_05: Naming of elements in Simulink / Stateflow models
Priority	Mandatory
Scope	Simulink / Stateflow
Prerequisites	None
Description	
	All elements in Simulink / Stateflow models that can be named must have a name.
Rationale	
	Finding matches between UML, Simulink and Stateflow models relies on naming.

Table-A III-26 Guideline av_us_06: Naming of Simulink Inport and Outport blocks.

Guideline (ID: Title)	av_us_06: Naming of Simulink Inport and Outport blocks
Priority	Recommended
Scope	Simulink
Prerequisites	
•	MAAB Style Guideline jc_0211: Usable characters for Inport block and Outport block
•	MAAB Style Guideline jm_0010: Port block names in Simulink models
Description	
	The names of Simulink Inport and Outport blocks must start with the prefixes *In_* and *Out_*, respectively.
Rationale	
	Finding matches between UML, Simulink and Stateflow models relies on naming.

Table-A III-27 Guideline av_us_07: Decomposition type for Stateflow chart and composite state.

Guideline (ID: Title)	av_us_07: Decomposition type for Stateflow chart and composite state
Priority	Recommended
Scope	Stateflow
Prerequisites	None
Description	
	A Stateflow chart or composite state should have a parallel (AND) decomposition and at least one parallel state owning a set of vertices (states and junctions) and transitions that define a behaviour fragment.
Rationale	
	A UML state machine and composite state denote a set of orthogonal behaviour fragments with the use of regions. Thus, this guideline is intended to reduce ambiguities when mapping a UML state machine or composite state to a Stateflow chart or composite state, respectively. The use of a parallel decomposition and parallel states is semantically equivalent to that of UML orthogonal regions. When the state machine or composite state define only one region this guideline can be elided.

Table-A III-28 Guideline av_us_08: Expression of triggers in UML transitions.

Guideline (ID: Title)	av_us_08: Expression of triggers in UML transitions
Priority	Mandatory
Scope	UML
Prerequisites	None
Description	
	UML triggers must be defined as event names denoted textually as described in subclause 13.3.4 of the UML 2.5.1 specification.
Rationale	
	This guideline is intended to allow a mapping to be established between a UML state machine and a Stateflow chart.
See Also	
•	UML 2.5.1 subclause 13.3.4 Notation

Table-A III-29 Guideline av_us_09: Expression of triggers in Stateflow transitions.

Guideline (ID: Title)	av_us_09: Expression of triggers in Stateflow transitions
Priority	Mandatory
Scope	Stateflow
Prerequisites	None
Description	
	Stateflow events in transitions should not be used for the reasons pointed out in Ferrari *et al.* (2013). An exception to the previous restriction applies to relative time event triggers, which must be denoted with "`after`" followed by the argument values corresponding to the number and unit of time.
Rationale	
	This guideline is intended to allow a mapping to be established between a UML state machine and a Stateflow chart.
See Also	
•	MAAB Guideline db_0126: Scope of events

Table-A III-30 Guideline av_us_10: Expression of triggers appearing in both UML and Stateflow transitions.

Guideline (ID: Title)	av_us_10: Expression of triggers appearing in both UML and Stateflow transitions
Priority	Mandatory
Scope	UML and Stateflow
Prerequisites	
•	Guideline av_us_08: Expression of triggers in UML transitions
•	Guideline av_us_09: Expression of triggers in Stateflow transitions
Description	
	If a transition appears in a UML state machine as well as a Stateflow chart, then the name of the Stateflow condition variable must be a subset of the UML trigger event name. An exception to the previous restriction applies to relative time event triggers, which must be denoted with "`after`" followed by the argument values corresponding to the number and unit of time.
Rationale	
	This guideline is intended to allow a mapping to be established between a UML state machine and a Stateflow chart.

Table-A III-31 Guideline av_us_11: Expression of UML guards in transitions.

Guideline (ID: Title)	av_us_11: Expression of UML guards in transitions
Priority	Mandatory
Scope	UML
Prerequisites	None
Description	
	UML guards in transitions must be defined as logical expressions made up of a primary expression, or a conjunction or disjunction of two or more primary expressions. A primary expression is defined to be a relational expression containing one relational operator (<, <=, >, >=, ~=, ==, ~).
Rationale	
	The UML specification defines a guard as an `OpaqueExpression`. Only certain constraints are predefined in UML. Thus, this guideline borrows the Stateflow notation for the expression of guards in UML to facilitate their mapping.
See Also	
•	UML 2.5.1 subclause 13.3.4 Notation (Triggers)
•	UML 2.5.1 subclause 7.6.4 Notation (`OpaqueExpression`)

Table-A III-32 Guideline av_us_12: Expression of Stateflow conditions in transitions.

Guideline (ID: Title)	av_us_12: Expression of Stateflow conditions in transitions
Priority	Mandatory
Scope	Stateflow
Prerequisites	MAAB Style Guideline db_0150: State machine patterns for conditions
Description	
	Stateflow conditions in transitions must be defined as logical expressions made up of a primary expression, or a conjunction or disjunction of two or more primary expressions. A primary expression is defined to be one of the following: • a Boolean input or local variable, or • a relational expression containing one relational operator (<, <=, >, >=, ~=, ==, ~).
Rationale	
	This guideline is intended to allow a mapping to be established between a UML state machine and a Stateflow chart.

Table-A III-33 Guideline av_us_13: Expression of UML actions.

Guideline (ID: Title)	av_us_13: Expression of UML actions
Priority	Mandatory
Scope	UML
Prerequisites	
•	MAAB Guideline jc_0501: Format of entries in a State block
•	MAAB Guideline db_0151: State machine patterns for transition actions
Description	
	UML actions on entry, do and exit behaviours for states and on transitions must be defined as sequences of UML Operation calls.
Rationale	
	The UML specification provides relatively minimal concrete syntax for actions in behaviours attached to states and transitions in a state machine. Thus, this guideline clarifies the type of behaviour expressions allowed. The guideline is also intended to facilitate the mapping of actions between UML state machines and Stateflow charts.
See Also	
•	UML 2.5.1 subclause 16.1.1.1 Concrete Syntax
•	UML 2.5.1 subclause 7.6.4 Notation (`OpaqueExpression`)

Table-A III-34 Guideline av_us_14: Expression of Stateflow actions.

Guideline (ID: Title)	av_us_14: Expression of Stateflow actions
Priority	Mandatory
Scope	Stateflow
Prerequisites	
•	MAAB Guideline jc_0501: Format of entries in a State block
•	MAAB Guideline db_0151: State machine patterns for transition actions
Description	
	Stateflow actions on entry, do and exit behaviours for states and on transitions must be defined as sequences of variable assignment statements. If the actions appear in a UML state machine as well, then the variable names must be subsets of the UML operation names used.
Rationale	
	The guideline is intended to facilitate the mapping of actions between UML state machines and Stateflow charts.
See Also	
•	UML 2.5.1 subclause 16.1.1.1 Concrete Syntax
•	UML 2.5.1 subclause 7.6.4 Notation (`OpaqueExpression`)

Table-A III-35 Guideline av_us_15: Expression of actions appearing in both UML and Stateflow models.

Guideline (ID: Title)	av_us_15: Expression of actions appearing in both UML and Stateflow models
Priority	Mandatory
Scope	UML and Stateflow
Prerequisites	
•	Guideline av_us_07: Expression of UML actions (see Table III-27)
•	Guideline av_us_08: Expression of Stateflow actions (see Table III-28)
Description	
	If an action appears in a UML state machine as well as a Stateflow chart, then the Stateflow variable names must be subsets of the UML operation names used.
Rationale	
	The guideline is intended to facilitate the mapping of actions between UML state machines and Stateflow charts.

Table-A III-36 Guideline av_us_16: Use of signal receipt and send symbols in UML state machines.

Guideline (ID: Title)	av_us_16: Use of signal receipt and send symbols in UML state machines
Priority	Recommended
Scope	UML
Prerequisites	None
Description	
	The UML signal receipt and send symbols should not be used. Instead, the UML default textual notation for a
Rationale	
	The signal receipt and send symbols break up the transition (into at least two segments), which makes more difficult a consistency analysis with other models (*e.g.*, Stateflow). Furthermore, these symbols are not always supported in tools.
See Also	
•	UML 2.5.1 subclause 14.2.4.8 Transition
•	Guideline av_us_08: Expression of triggers in UML transitions (see Table III-28)
•	Guideline av_us_13: Expression of UML actions (see Table III-33)

Table-A III-37 Guideline av_us_17: Use of UML fork and join pseudostates.

Guideline (ID: Title)	av_us_17: Use of UML fork and join pseudostates
Priority	Mandatory
Scope	UML
Prerequisites	Guideline av_us_04: Naming of UML fork and join pseudostates (Table III-24)
Description	
	When using fork and join pseudostates, every parallel path from the fork pseudostate to the join pseudostate has only one composite state.
Rationale	
	This guideline is intended to reduce ambiguities when mapping a UML state machine and a Stateflow chart.

Table-A III-38 Guideline av_us_18: Data type of Simulink Inports and Outports.

Guideline (ID: Title)	av_us_18: Data type of Simulink Inports and Outports
Priority	Mandatory
Scope	Simulink
Prerequisites	None
Description	
	The data type of Simulink Inports and Outports must be set. The data type cannot be inherited or auto.
Rationale	
	Finding matches between UML, Simulink and Stateflow models relies on data types.

Table-A III-39 Guideline av_us_19: Conjugation of a UML Port.

Guideline (ID: Title)	av_us_19: Conjugation of a UML Port
Priority	Mandatory
Scope	UML
Prerequisites	None
Description	
•	The value of the `isConjugated` property of a UML port must be set accordingly depending on the port's purpose. • If the port is used to provide an interface, then the value of the `isConjugated` property must be set to `false`. • If the port is used to require an interface, then the value of the `isConjugated` property must be set to `true`.
Rationale	
•	The value of the `isConjugated` property specifies the way that the provided and required interfaces are derived from the port's type.
See Also	
•	UML 2.5.1 subclause 11.8.14 Port

Table-A III-40 Guideline av_us_20: Entry and exit points in Stateflow chart.

Guideline (ID: Title)	av_us_20: Entry and exit points in Stateflow chart
Priority	Recommended
Scope	Stateflow
Prerequisites	None
Description	
	In some modelling situations when encapsulating elements in a composite state, it is useful to bind the internal elements of such composite state with incoming and outgoing transitions. In UML this is realized by means of entry and exit point pseudostates. Stateflow does not provide equivalent constructs for such purpose. However, incoming transitions can penetrate a composite state and terminate directly on one of its internal vertices, and outgoing transitions can start in an internal state and terminate in another state outside the composite state. When these situations need to be modelled in a Stateflow chart an additional state should be added within the composite state representing an entry or exit point of the composite state. For the case of exit points, the entry action of the additional state must set a local data in the Stateflow chart that is evaluated in the conditions of the state's corresponding outgoing transitions.
Rationale	
	This guideline is intended to reduce ambiguities when mapping a UML state machine and a Stateflow chart. The introduction of additional states into the Stateflow chart to represent entry and exit points of composite states can be regarded as semantically equivalent to that of UML entry and exit point pseudostates.

Table-A III-41 Guideline av_us_21: Fork and join behaviour in Stateflow.

Guideline (ID: Title)	av_us_21: Fork and join behaviour in Stateflow
Priority	Recommended
Scope	Stateflow
Prerequisites	None
Description	
	In some modelling situations where all parallel states need to terminate their execution before the execution can continue, the parallel states must perform a synchronization. To perform a synchronization function in Stateflow parallel states, additional states are required as follows: • An additional 'Synced' state within each parallel state must be added as the final state. • The previous 'Synced' state must be reached through a transition that evaluates to 'true' a Boolean 'syncing' local data. When the state is reached, the Boolean 'synced' local data is changed to 'true'. • One of the parallel states needs to be chosen as the synchronizing state. Such a state will change the 'syncing' local data to 'true' in the transition to an additional 'Syncing' state placed before the 'Synced' state.
Rationale	
	Stateflow parallel states do not have a synchronization mechanism. Thus, this guideline is intended to define a synchronization mechanism. A word of caution, synchronizing parallel states in this manner is not scalable and is recommended only for synchronizing two parallel states as it could cause state explosion.
See Also	
•	Stateflow documentation: Broadcast Events to Synchronize States

APPENDIX IV

CHECSDM AND *CHECSDM4USS* DEVELOPER'S AND USER'S GUIDES

1. Developer's Guide

1.1 Getting started

System requirements

- Java JDK 8 or higher
- Eclipse Oxygen or newer with the Modeling package (*a.k.a.* Eclipse Modeling Tools)
- MATLAB R2018b

Install required dependencies

The required dependencies are Epsilon, Viatra and Xtext.

1. Go to **Help** » **Install New Software...**.
2. Work with http://download.eclipse.org/epsilon/1.4/updates/ to install Epsilon. Select the following features:
   - Epsilon Core v1.4.0
   - Epsilon Core Development Tools v1.4.0
   - Epsilon EMF Integration v1.4.0
   - Epsilon UML Integration v1.4.0
3. Work with http://download.eclipse.org/modeling/tmf/xtext/updates/composite/releases/ to install Xtext. Select the following features:

- Xtext » Xtext Complete SDK v2.12.0

4. Work with http://download.eclipse.org/viatra/updates/release to install Viatra. Select the following features:

   - VIATRA Core » VIATRA Query and Transformation SDK v1.6.1

Import the *checsdm4uss* projects

1. Import the source code projects of *checsdm4uss* into the Eclipse workspace.

Create new run configuration

Create a new Run Configuration in Eclipse:

1. Go to **Run** » **Run Configurations...**.
2. From the list in the left, double click on **Eclipse Application**.
3. Give the configuration a name (suggestion: **checsdm4uss**).
4. Select the tab **Environment**.
5. Create a new variable with the following information:

   In Windows:

   Name: PATH

   Value: `«path to MATLAB»\bin\win64`

   In macOS:

   Name: DYLD_LIBRARY_PATH

   Value: `«path to MATLAB»/bin/maci64`

In Linux:

Name: LD_LIBRARY_PATH

Value: «path to MATLAB»/bin/glnxa64:«path to MATLAB»/sys/os/glnxa64

1.2 Project structure

- **src-gen**

  This folder contains generated source code from models. These files should not be edited directly.

- **src**

  This folder contains manually created source code. These files can be edited directly if necessary.

- **icons**

  This folder contains image files used for icons in the plug-in.

- **META-INF** and **MANIFEST.MF**

  This folder and file contain descriptions of the extensions made to Eclipse IDE by the plug-in.

- **models**

  This folder contains EMF models.

- **build.properties**

  This file contains the build properties of the plug-in.

- **plugin.properties**

  This file contains additional properties of the plug-in.

- **plugin.xml**

  This file contains descriptions of the extensions made to the Eclipse IDE by the plug-in.

1.3 *checsdm4uss* projects and noteworthy files

- **ca.ets.sofeess.checsdm4uss.guidelinecompliance**

  This project contains the guideline formalizations in VQL. Each .vql file contains one guideline formalization. The files are named with the guideline IDs.

- **ca.ets.sofeess.checsdm4uss.guidelinecompliance.validation**

  This project is automatically generated by Viatra. It contains all the marker-based validation code generated from the constraints specified with the `@Constraint` annotation in the guideline formalizations.

- **ca.ets.sofeess.checsdm4uss.mappingrules**

  This project contains the mapping rules formalizations in ECL.

  – **Main.ecl**

    Entry point for mapping rule execution. This file contains initialize (`pre`) and terminate (`post`) code blocks that must be executed before and after (respectively) evaluating the mapping rules.

  – **Common.ecl**

    This file defines common operations used throughout the mapping rules formalizations.

  – **ClassesAndSubsystems.ecl**

    This file defines the formalizations for mapping rules regarding UML components and classes, and Simulink subsystem blocks, *i.e.* mapping rules mr_us_01 through mr_us_-08 (see Appendix III).

  – **StateMachinesAndCharts.ecl**

    This file defines the formalizations for mapping rules regarding UML state machines and Stateflow charts, *i.e.* mapping rules mr_us_09 through mr_us_20 (see Appendix III).

- **ca.ets.sofeess.checsdm4uss.mappingrules.ui**

  This project provides a contextual menu UI contribution to the Eclipse IDE for executing the mapping rules when both a UML and a Simulink model are selected in the *Model Explorer* view.

- **ca.ets.sofeess.checsdm.mappings**

  This project defines the Mapping metamodel.

  – **Mappings.ecore**

    EMF model defining the Mapping metamodel (see Figure 3.4).

  – **Mappings.genmodel**

    EMF generator model containing the control parameters on how code and other derived outputs should be generated.

- **ca.ets.sofeess.checsdm.mappings.edit**

  This project provides editing facilities for mapping models. This project is generated through the *Mappings.genmodel* generator model in the *ca.ets.sofeess.checsdm.mappings* project.

- **ca.ets.sofeess.checsdm.mappings.editor**

  This project provides a graphical editor for mapping models. This project is generated through the *Mappings.genmodel* generator model in the *ca.ets.sofeess.checsdm.mappings* project.

- **ca.ets.sofeess.breesse.engine**

  This project provides a high-level, reusable, extendable framework for working programmatically with Simulink and Stateflow through MATLAB's Java API.

- **ca.ets.sofeess.breesse.simulink**

  This project defines the Simulink EMF metamodel.

  - **Simulink.ecore**

    EMF model defining the Simulink EMF metamodel.

  - **Simulink.genmodel**

    EMF generator model containing the control parameters on how code and other derived outputs should be generated.

- **ca.ets.sofeess.breesse.simulink.edit**

  This project provides editing facilities for Simulink EMF models. This project is generated through the *Simulink.genmodel* generator model in the *ca.ets.sofeess.checsdm.simulink* project.

- **ca.ets.sofeess.breesse.simulink.editor**

  This project provides a graphical editor for Simulink EMF models. This project is generated through the *Simulink.genmodel* generator model in the *ca.ets.sofeess.checsdm.simulink* project.

- **ca.ets.sofeess.breesse.importer.api**

  This project provides facilities for importing Simulink.slx files into the Eclipse IDE.

- **ca.ets.sofeess.breesse.importer.ui**

  This project provides a contextual menu UI contribution to the Eclipse IDE for importing a Simulink.slx file selected in the *Model Explorer* view.

- **matlabengine.bundle**

  This project bundles the MATLAB Java API as an Eclipse plug-in.

  - **engine.jar**

    The MATLAB Java API file. The current file corresponds to MATLAB R2018b. If the installed MATLAB release differs, replace the file by importing the one found in «path to MATLAB»/extern/engines/java/jar/engine.jar.

- **org.apache.commons.text**

  This project bundles the Apache Commons Text library as an Eclipse plug-in. This library provides algorithms working on strings. Some of these algorithms are used when comparing names in the mapping rules formalizations.

  – **commons-text-1.1.jar**

    The Apache Commons Text library. The library can be updated by replacing the file with a more recent version.

1.4 More information

Viatra

More information about Viatra is available at https://www.eclipse.org/viatra/documentation/index.html.

ECL

More information about ECL is available in Kolovos *et al.* (2018).

MATLAB

More information about MATLAB, Simulink and Stateflow is available at https://www.mathworks.com/help/simulink/.

2. User's Guide

2.1 Getting started

System requirements

- Java JDK 8 or higher

- Eclipse Oxygen or newer with the Modeling package (*a.k.a.* Eclipse Modeling Tools)
- MATLAB R2018b

Before running

Create a new environment variable in your operating system with the following information:

In Windows:

Name: PATH

Value: `«path to MATLAB»\bin\win64` **Note:** If a PATH variable already exists, append the value to it. Make it the first value of the variable. You must restart the computer afterward before continuing, otherwise the Simulink Importer will not work.

In macOS:

Name: DYLD_LIBRARY_PATH

Value: `«path to MATLAB»/bin/maci64`

In Linux:

Name: LD_LIBRARY_PATH

Value: `«path to MATLAB»/bin/glnxa64:«path to MATLAB»/sys/os/glnxa64`

Install required dependencies

The required dependencies are Epsilon, Viatra and Xtext.

1. Go to **Help** » **Install New Software...**.

2. Work with http://download.eclipse.org/epsilon/1.4/updates/ to install Epsilon. Select the following features:
   - Epsilon Core v1.4.0
   - Epsilon Core Development Tools v1.4.0
   - Epsilon EMF Integration v1.4.0
   - Epsilon UML Integration v1.4.0
3. Work with http://download.eclipse.org/modeling/tmf/xtext/updates/composite/releases/ to install Xtext. Select the following features:
   - Xtext » Xtext Complete SDK v2.12.0
4. Work with http://download.eclipse.org/viatra/updates/release to install Viatra. Select the following features:
   - VIATRA Core » VIATRA Query and Transformation SDK v1.6.1

Install the *checsdm4uss* plug-ins

1. Place the *checsdm4uss* plug-ins under «`path to Eclipse `»`/plugins`.
2. (Re)Start the Eclipse IDE.

***checsdm4uss* plug-ins**

- ca.ets.sofeess.checsdm4uss.guidelinecompliance_1.0.0.jar
- ca.ets.sofeess.checsdm4uss.guidelinecompliance.validation_1.0.0.jar
- ca.ets.sofeess.checsdm4uss.mappingrules.ui_1.0.0.jar
- ca.ets.sofeess.checsdm.mappings_1.0.0.jar

- ca.ets.sofeess.checsdm.mappings.edit_1.0.0.jar
- ca.ets.sofeess.checsdm.mappings.editor_1.0.0.jar
- ca.ets.sofeess.breesse.engine_1.0.0.jar
- ca.ets.sofeess.breesse.simulink_1.0.0.jar
- ca.ets.sofeess.breesse.simulink.edit_1.0.0.jar
- ca.ets.sofeess.breesse.simulink.editor_1.0.0.jar
- ca.ets.sofeess.breesse.importer.api_1.0.0.jar
- ca.ets.sofeess.breesse.importer.ui_1.0.0.jar
- matlabengine.bundle_1.0.0.jar
- org.apache.commons.text_1.0.0.jar

Import the *checsdm4uss* mapping rules project

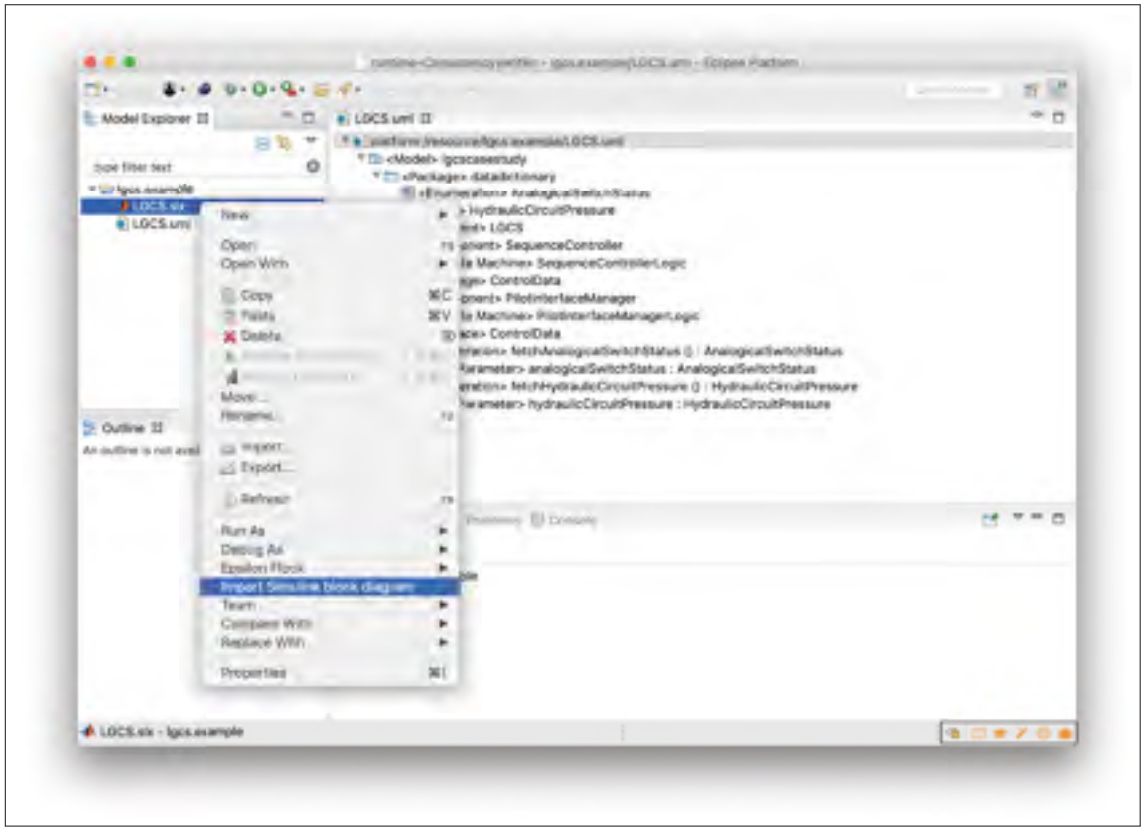
1. Import the **ca.ets.sofeess.checsdm.mappingrules** project into the Eclipse workspace.

2.2 Importing a Simulink model into Eclipse

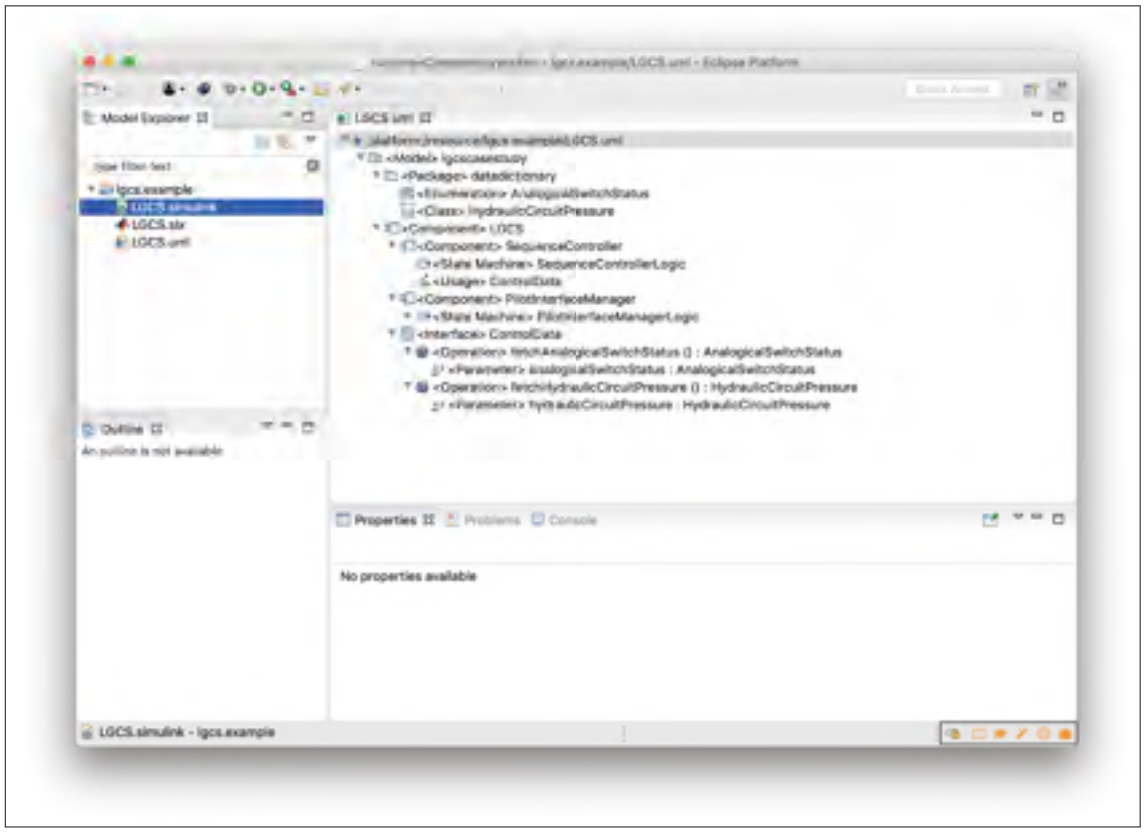
The following steps guide the import of a Simulink model into the Eclipse workspace.

1. Open the Eclipse IDE.
2. Create a project.
3. Move or copy and paste the .slx Simulink model file into the project.
4. Select the .slx file.

5. Right click on the .slx file and select **Import Simulink block diagram**.



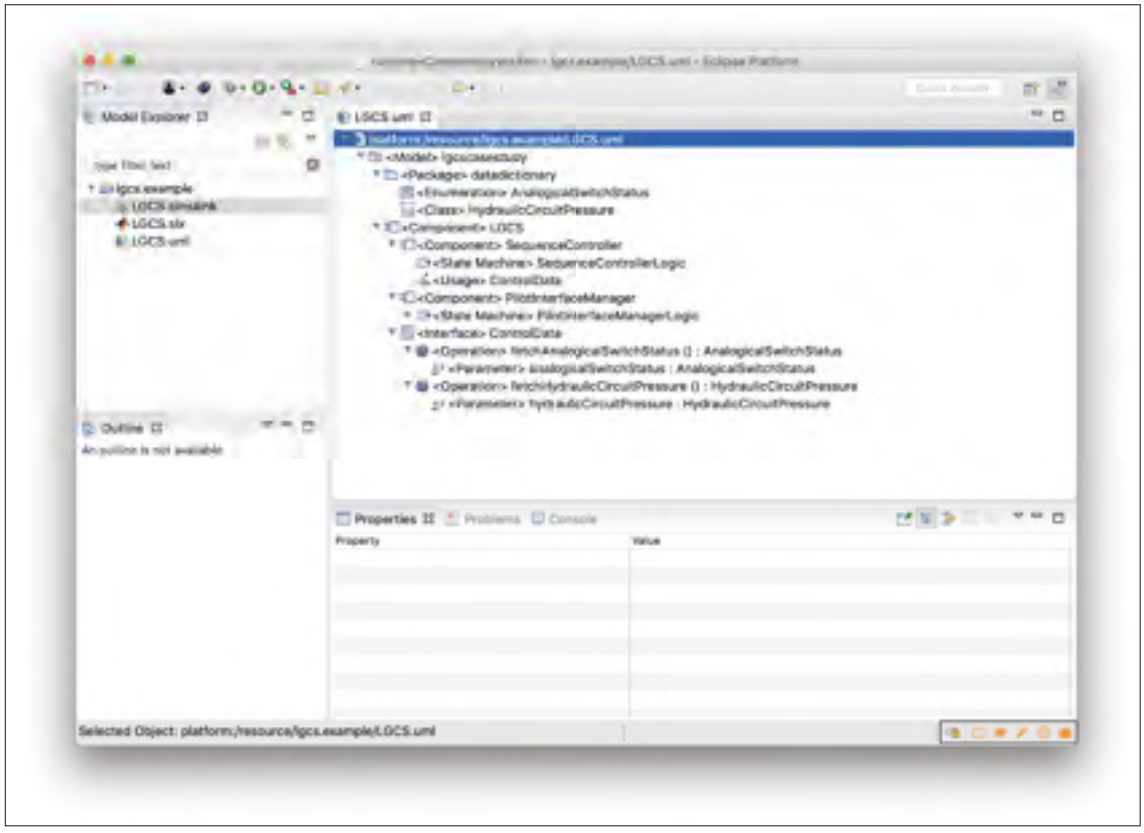
6. The import process starts. The import process may take a few minutes. MATLAB may pop up. This is the normal behaviour. Once the import process ends the MATLAB window will close and the EMF Simulink model should appear inside the project. If the block diagram makes use of library files, a dialog may pop up asking if these should be imported. A dialog may pop up if library models are used but could not be imported.



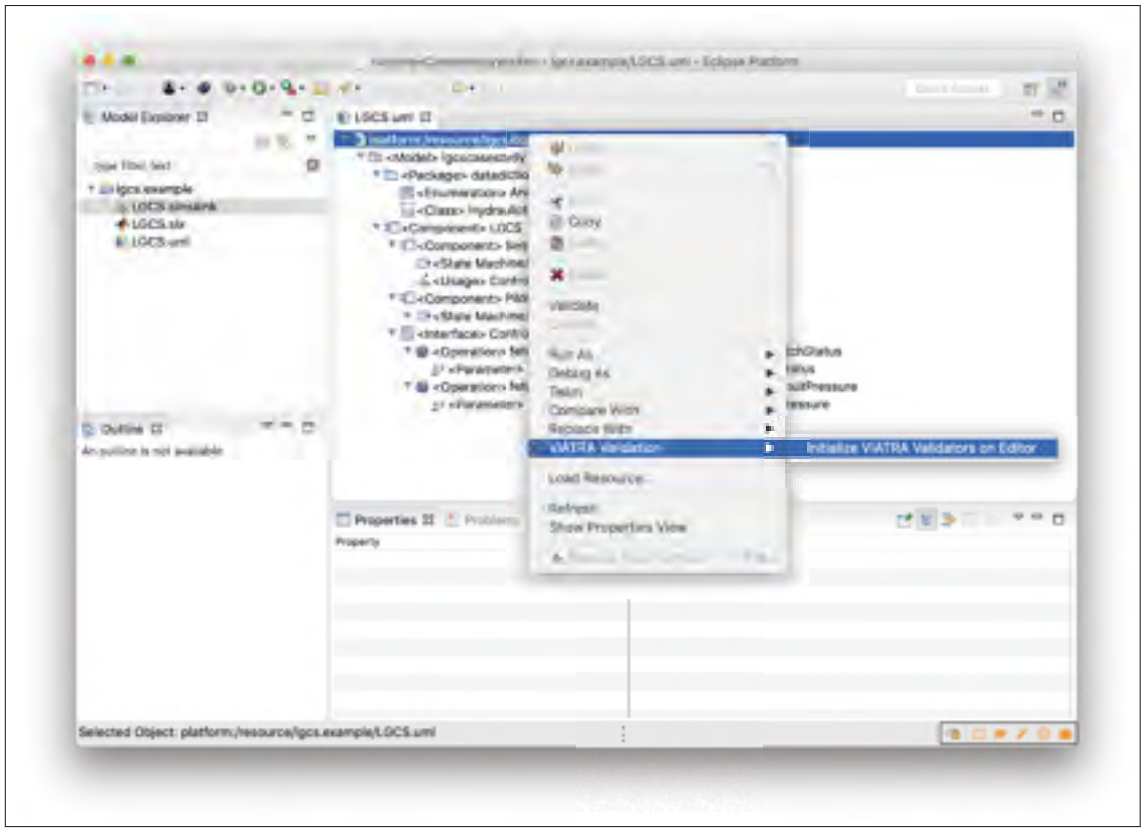
2.3 Verify guideline compliance

The following steps guide the verification of guideline compliance on any of the supported models (*i.e.* UML, Simulink/Stateflow, mapping model).

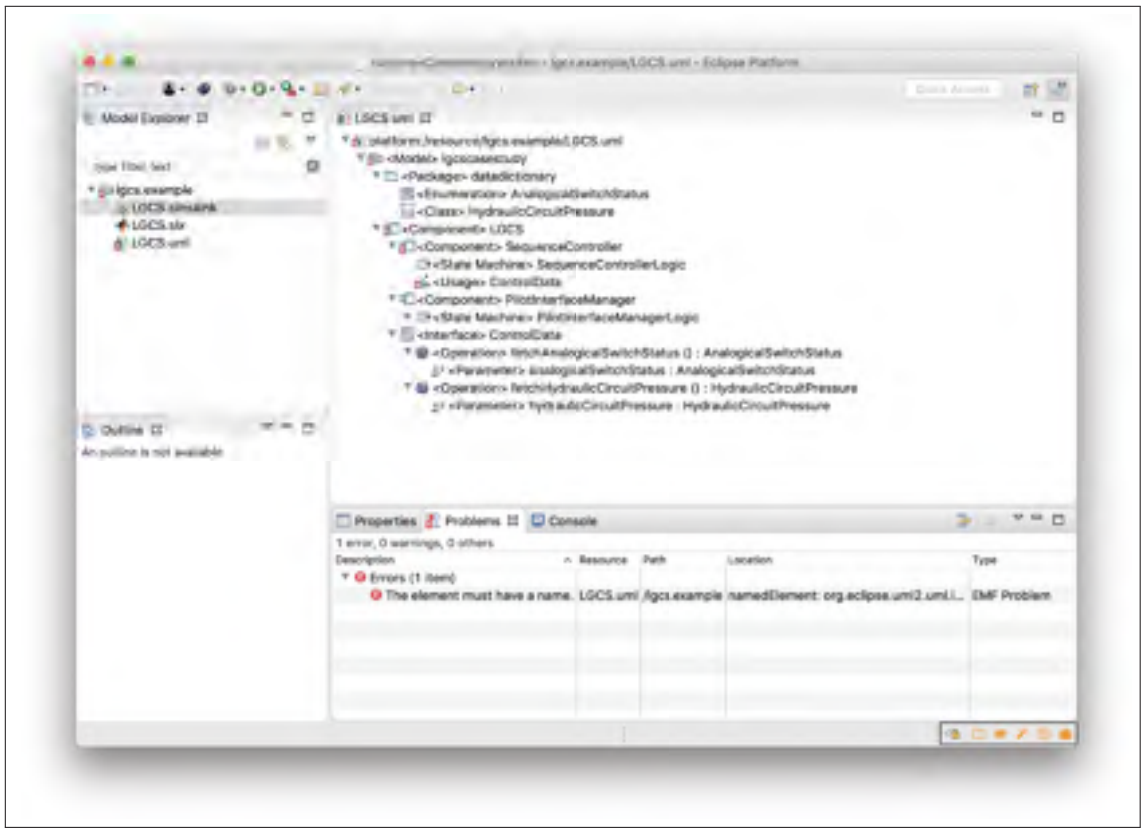
1. Open the model to be verified for guideline compliance. In this case, a UML model.



2. The verification can be carried out at anytime during modelling. Right click anywhere in the model and select **VIATRA Validation** » **Initialize VIATRA Validators on Editor**.



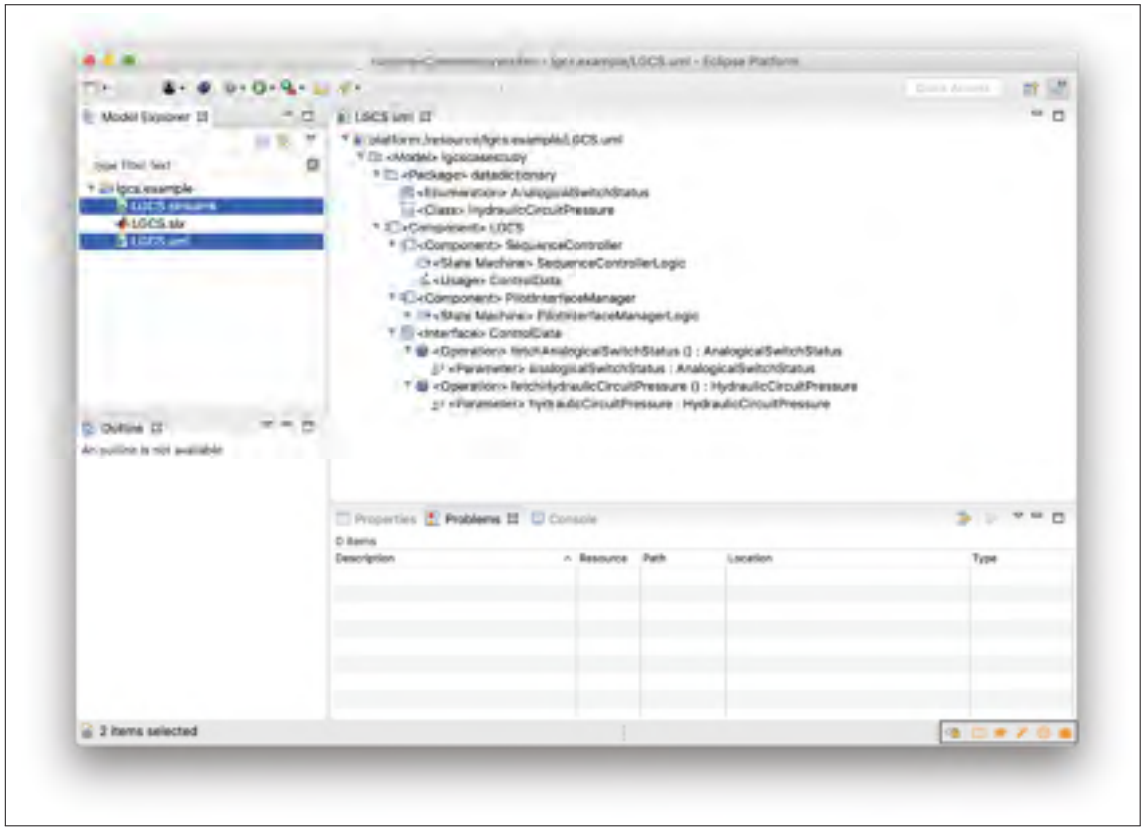
3. Violations will be presented in the **Problems** view (bottom of the screen). Any violating element will be marked in the editor view as well. To fix a violation, follow the indications given in the violation message.



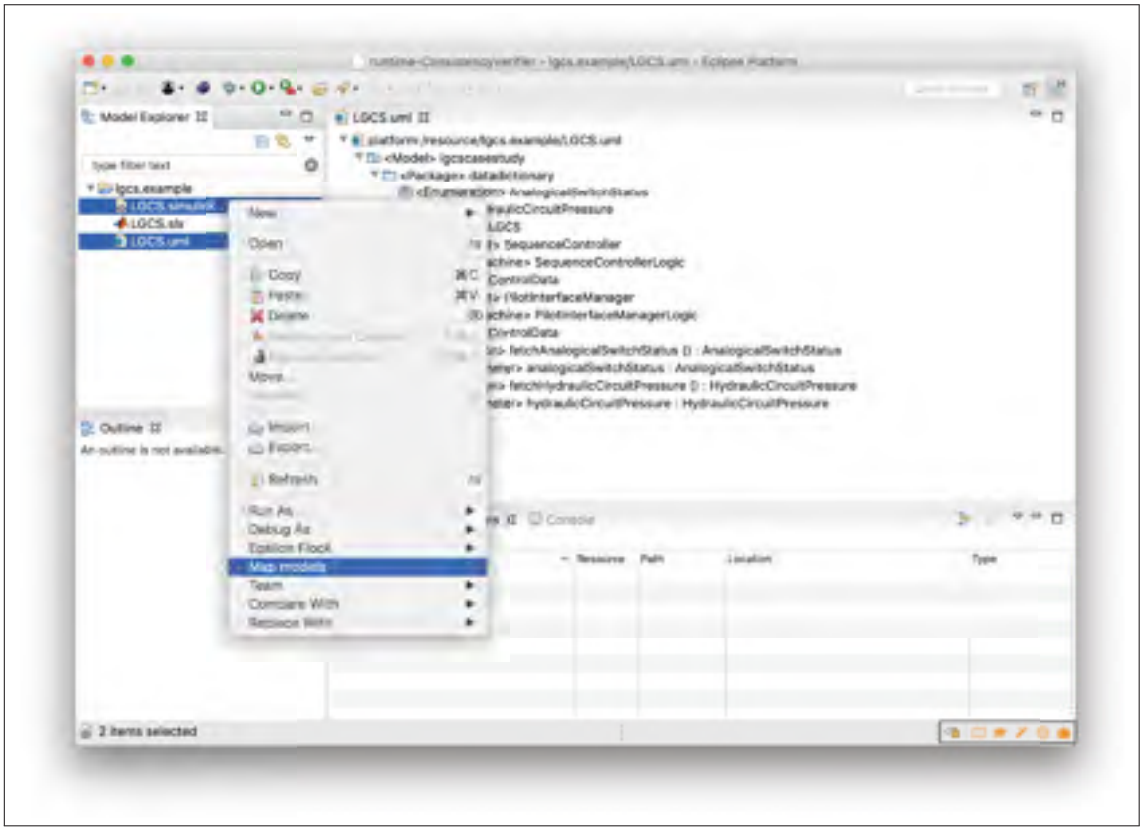
2.4 Map models

The following steps guide the mapping of design models.

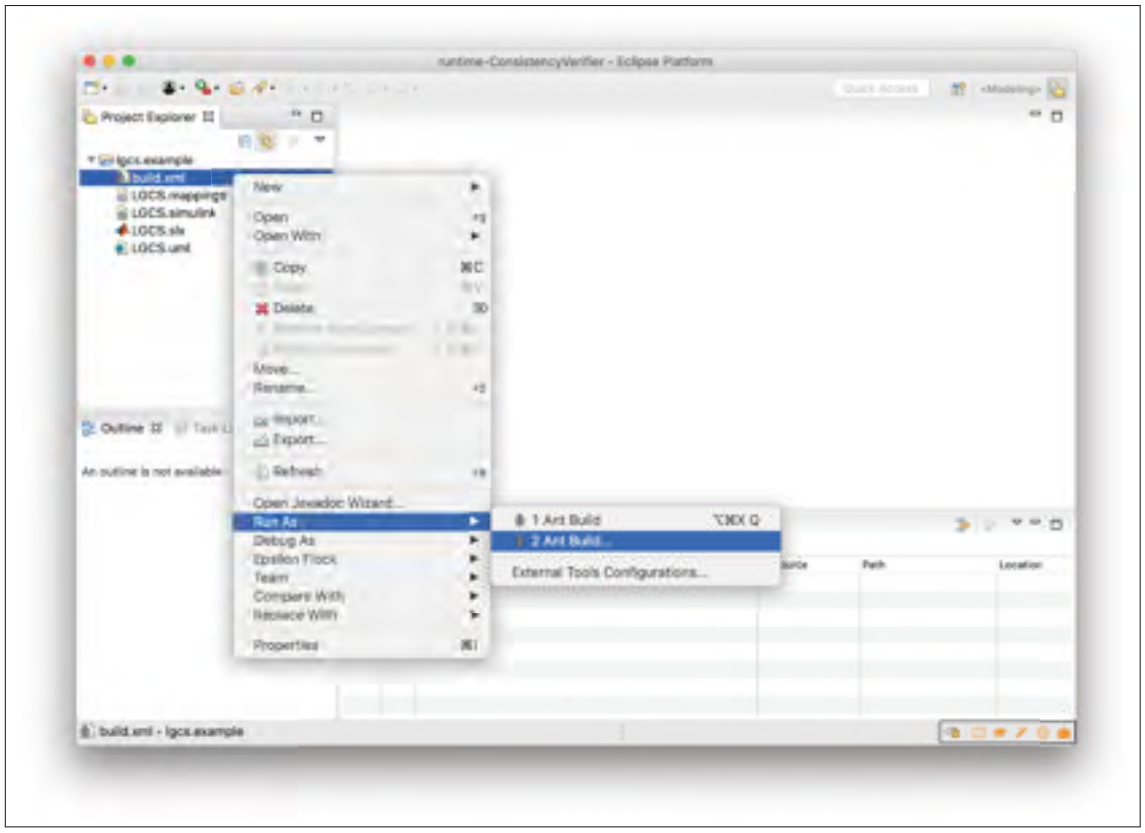
1. Select two models to map. In this case a Simulink EMF and UML models.



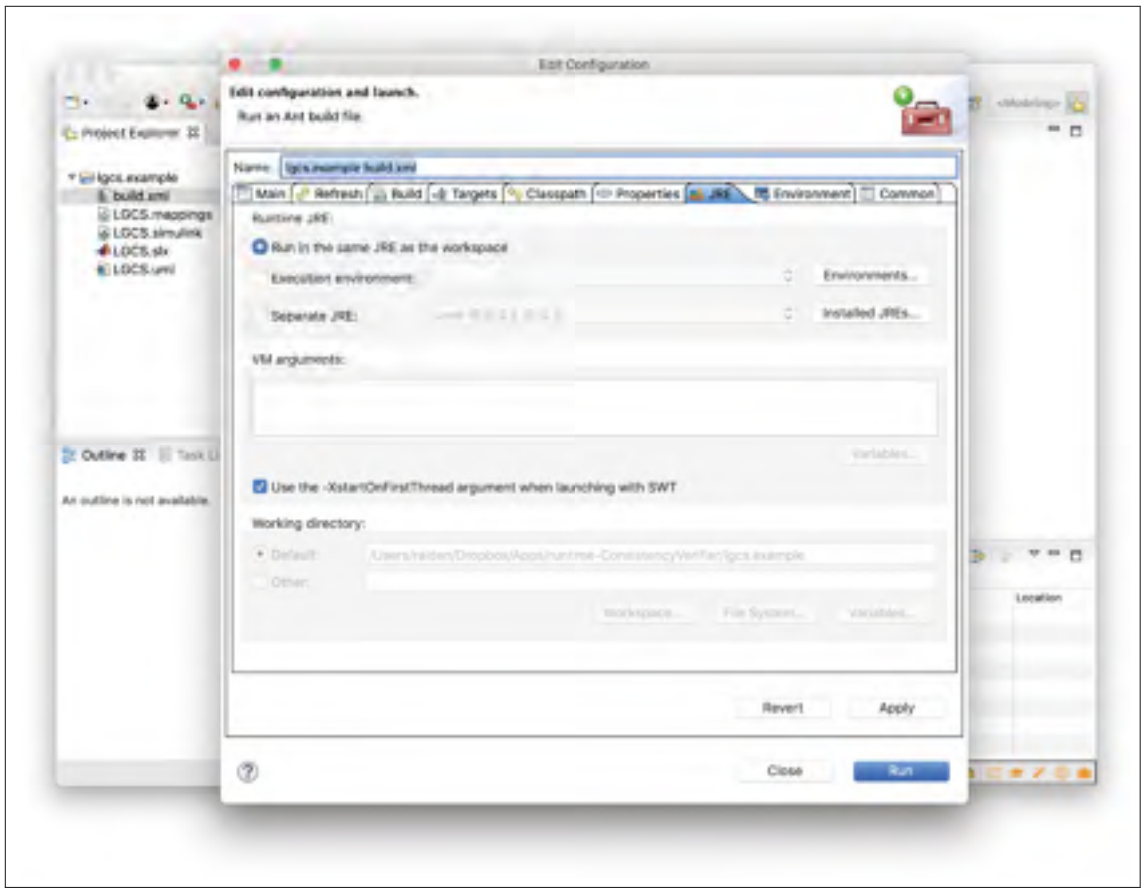
2. Right click over one of the selected models and select **Map models**.



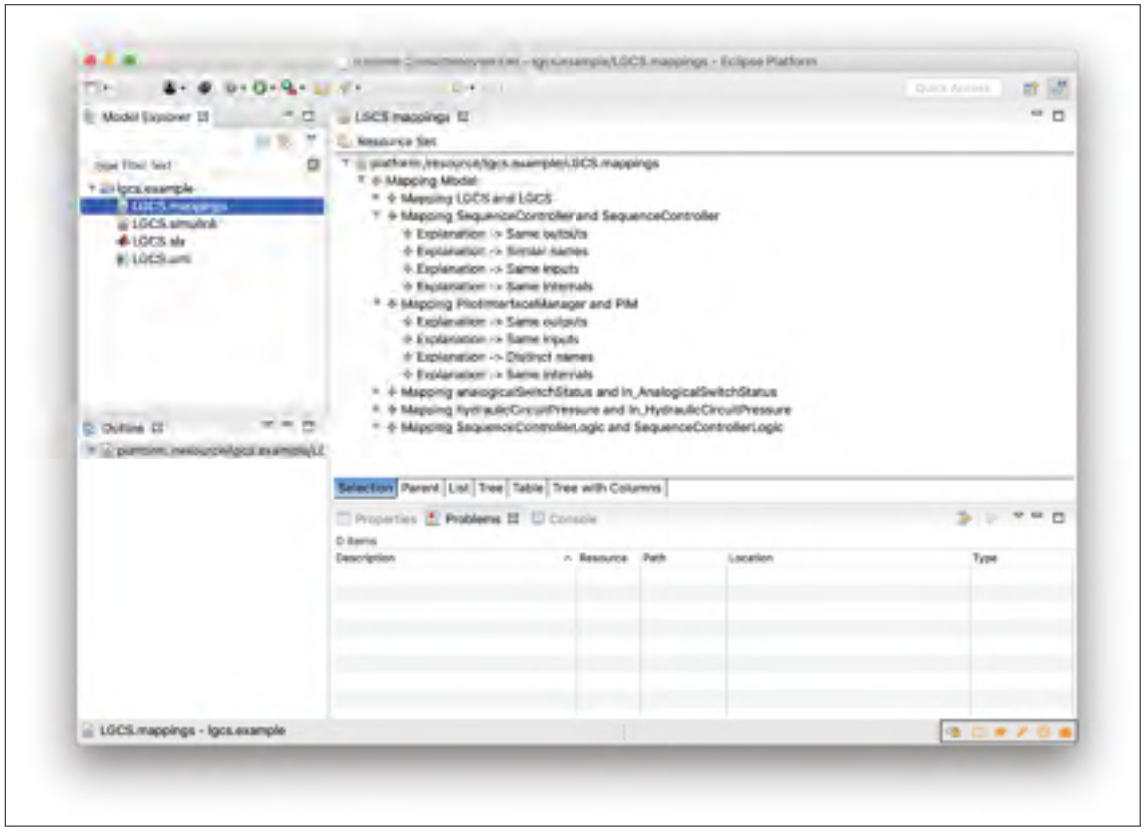
3. If the mapping fails, refresh the project and right click over the **build.xml** file that was generated inside the project. Go to **Run As** » **Ant Build...**.



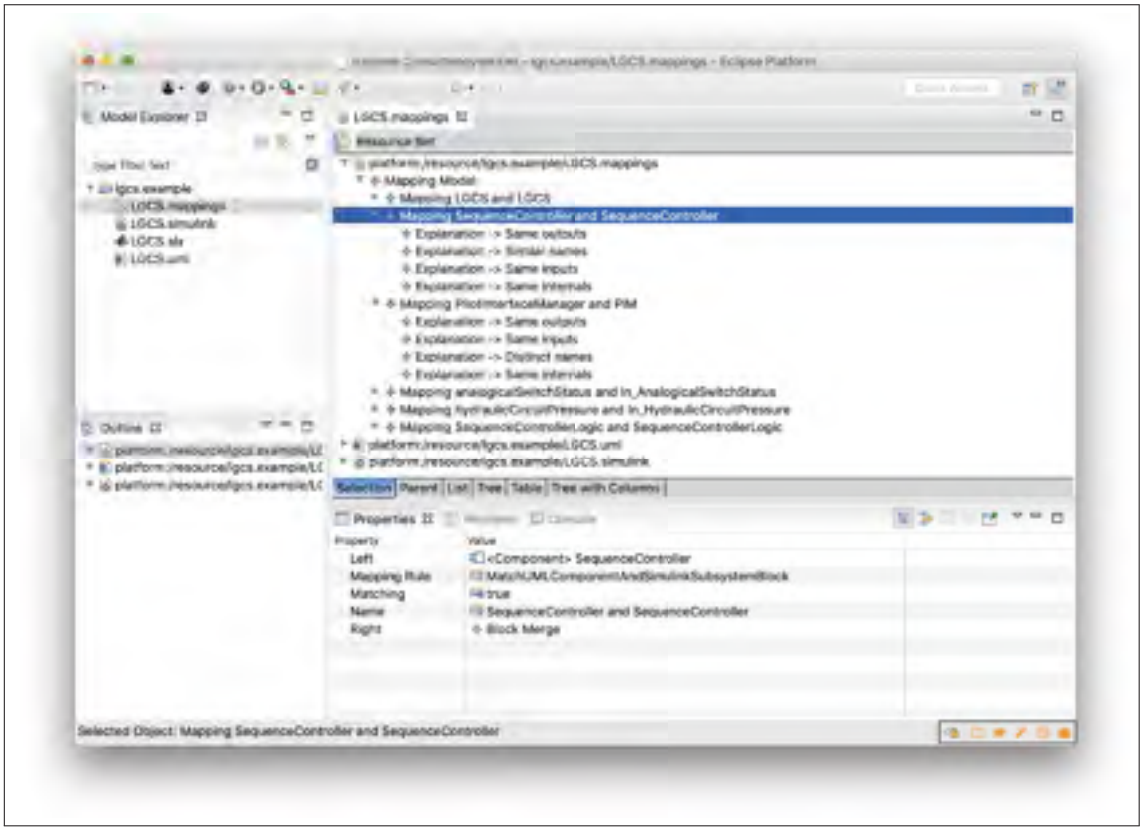
4. Select the **JRE** tab. Make sure the **Runtime JRE** is set to **Run in the same JRE as the workspace**. Click on **Apply** and then **Run**. This will rerun the mapping models operation.



5. Refresh the project. The mapping model should appear inside the project.



6. Select a mapping to review its details in the **Properties** view (bottom of the screen).



APPENDIX V

BREEZE THROUGH SAFETY-CRITICAL SYSTEM MODEL-BASED DESIGN WITH EMF, SIMULINK AND STATEFLOW

The following subsections highlight *Breesse*'s main features and describe its architecture. Two most notable elements in the architecture are further described: 1) the EMF-based Simulink and Stateflow metamodel, and 2) the command evaluator.

1. Overview

Breesse is a live, generic bridge for the EMF ecosystem and the MathWorks Simulink and Stateflow ecosystem. The bridge has been realized in pure Java with EMF technologies and packaged as an Eclipse plug-in for an easy deployment. Eclipse UI integration is also provided. *Breesse* makes use of the MATLAB Engine API for Java to directly connect to a running MATLAB instance, or initiate one. Figure-A V-1 shows the technology stack used to develop *Breesse*. *Breesse* is able to import the contents of Simulink and Stateflow design models and libraries into EMF-based Simulink and Stateflow representations. These EMF-based representations are straightforward and enable the manipulation of the design models in other existing EMF-based tools for MDE. A screenshot of *Breesse* in action is presented later in Section 5. The current release and complete source code of *Breesse* are available at Paz & El Boussaidi (2019a).

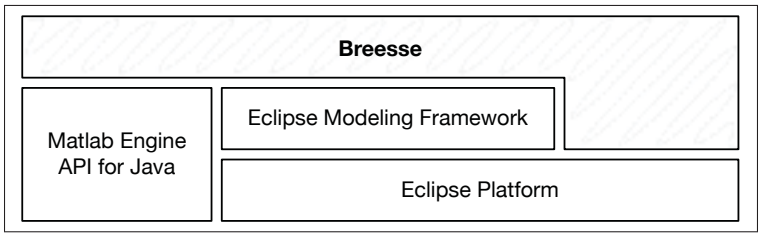


Figure-A V-1 Technology stack supporting *Breesse*.

2. Architecture

Figure-A V-2 presents the high-level component structure of *Breesse* and their interactions with the MATLAB Engine API for Java. At the core of *Breesse* is the EMF-based Simulink and Stateflow metamodel to which imported design models and libraries conform. The following subsection gives the details of such metamodel. The Simulink and Stateflow importer component provides facilities to import the contents of Simulink and Stateflow design models and libraries and create an EMF-based model conforming to the EMF-based Simulink and Stateflow metamodel. In order to create a generic bridge, able to support a wide range of operations, a command evaluator component was introduced. This component offers a set of high-level operations for extracting information from Simulink and Stateflow design models in MATLAB. These operations are translated into MATLAB Engine commands and sent for their execution using the MATLAB Engine API for Java.

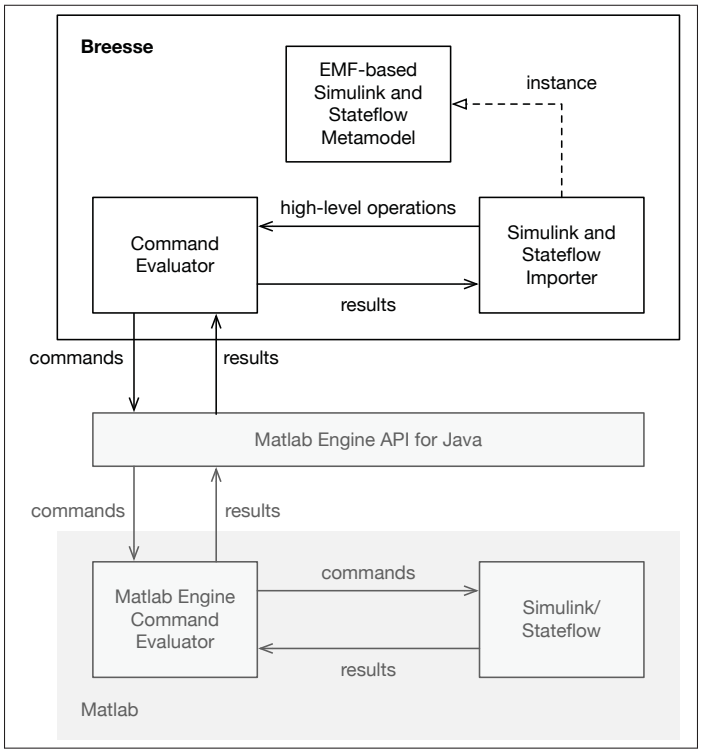


Figure-A V-2 *Breesse*'s high-level component structure and interactions.

3. The EMF-based Simulink and Stateflow Metamodel

The metamodel describes the concepts, and their relationships, found in both MathWorks Simulink and Stateflow. It is primarily based on Massif's EMF Simulink metamodel (IncQuery Labs, 2017). The complete metamodel is not shown, however, it is available online (Paz & El Boussaidi, 2019a). Figure-A V-3 shows a simplified excerpt of the main Simulink concepts. Shaded concepts are abstract concepts. All concepts extend the abstract concept SimulinkElement. This inheritance provides the common attributes in all the other concepts. Simulink contains Blocks that own a set of Ports. A Block can be a (predefined or customized) primitive block or typed as a logical structuring block such as a Subsystem—or one of its subtypes: SimulinkModel or Reference—or as a Stateflow Chart or TruthTableChart. Subsystems are regarded as logical structuring blocks since they solely impose a hierarchy of blocks and do not determine system behaviour. SimulinkModels are much the same except that their contents reside in a separate model file. Reference blocks are pointers to concrete reusable blocks defined in custom libraries. A Port is typed as either an InPort or an OutPort depending if it serves to receive or supply data, respectively. Ports can be connected between each other; with a connection going from an OutPort toward an InPort.

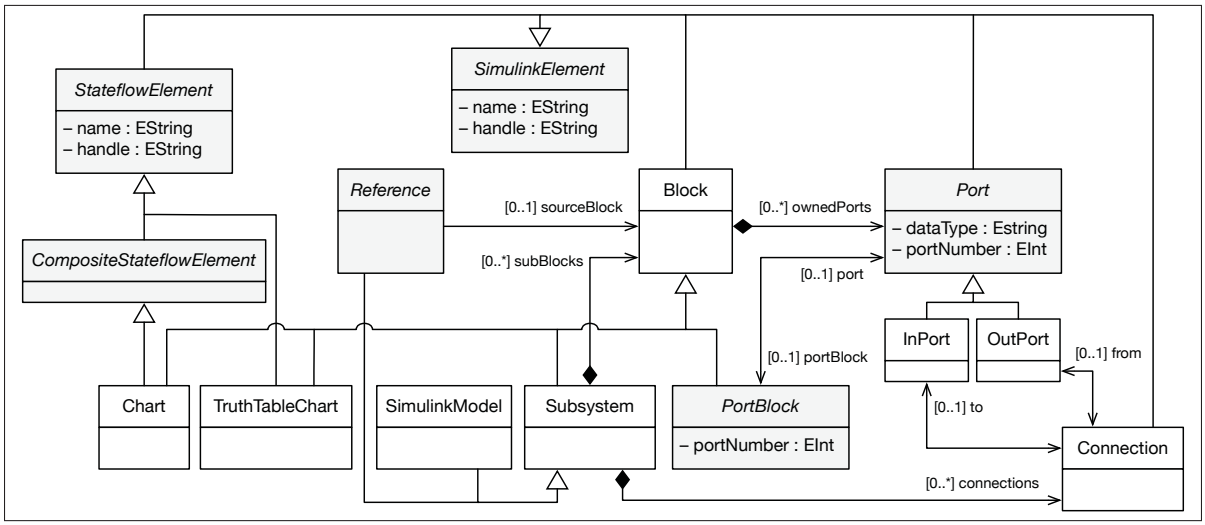


Figure-A V-3 Excerpt of the main Simulink concepts in the EMF-based Simulink and Stateflow metamodel, adapted from IncQuery Labs (2017).

Support for Stateflow is a major feature of *Breesse*. Figure-A V-4 presents a simplified excerpt of the main Stateflow concepts. All Stateflow concepts extend the abstract StateflowElement. Like with the *SimulinkElement* abstract concept, this inheritance gathers the shared attributes in all the Stateflow concepts. CompositeStateflowElements are those that comprise other elements to represent part-whole hierarchies. A CompositeStateflowElement can be typed as a Chart, *i.e.* a representation of a finite state machine, or a State, *i.e.* an operating mode within a finite state machine. In the latter, the State is regarded as a composite state. ContainableStateflowElements represent elements that can be children of CompositeStateflowElements. A ContainableStateflowElement can be typed as a Vertex (either Junction or State), Transition or a ContainableTruthTable. Transitions connect a source Vertex to a destination Vertex. A Transition without a specified source is interpreted as the *default transition*, *i.e.* the first transition taken when a CompositeStateflowElement is entered. Thus, this attribute is marked as a derived attribute. TruthTables specify combinatorial logic. These can be typed as ContainableTruthTable or TruthTableChart, whenever the truth table is contained within a CompositeStateflowElement or it is its own standalone Simulink block.

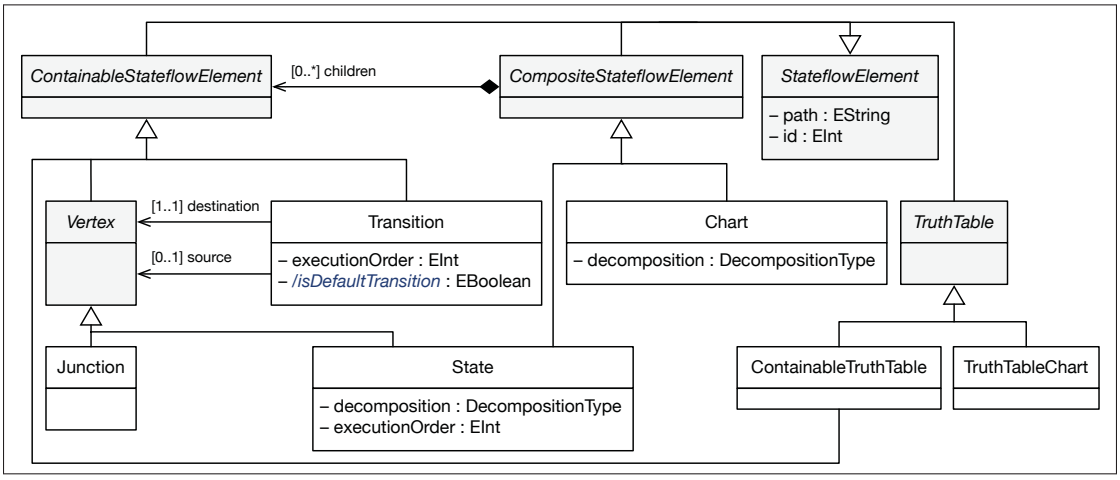


Figure-A V-4 Excerpt of Stateflow concepts in the EMF-based Simulink and Stateflow metamodel.

Design engineers can create Simulink and Stateflow design models using the metamodel. An EMF-based tree editor is provided for this purpose.

4. The Command Evaluator

The MATLAB Engine API for Java enables the interaction with MATLAB from Java programs. Key features of this API include: 1) starting and terminating a MATLAB instance, and 2) evaluating MATLAB functions/commands with the input arguments passed from within the Java program and receiving the output variables returned from MATLAB. Interaction with MATLAB via the MATLAB Engine API for Java, however, is based entirely on the expression of commands/functions as simple strings without any validation done at compile time or even content assist support. The command evaluator, hence, provides a framework for integrating MATLAB command/function evaluation capabilities directly into the Java language. MATLAB commands are written against strongly typed command objects using Java's familiar syntax. It is worth mentioning that the command evaluator framework is reusable on its own in other application contexts that involve interaction with MATLAB from a Java program.

The framework divides the interaction process into four parts: 1) engine management, 2) command specification/creation, 3) command evaluation, and 4) command result retrieval. In the first part, *engine management*, a *MatlabEngineManager* class hides the logic for starting and stopping MATLAB instances. *Command specification/creation* is supported by the use of the *factory* and *builder* design patterns. Figures-A V-5 and V-6 show sample structural views of their applications, respectively. The `MatlabCommandFactory` encapsulates the commands' initial creation logic in simple-to-use methods that return the required instance. Passing arguments to a MATLAB function/command can be cumbersome since they sometimes require appending keywords. Furthermore, depending on the expected result, the amount of arguments varies. `MatlabCommands` enable themselves a step by step construction of the command object. The bottom part of Figure V-6 depicts the specification of a command to find all Stateflow charts in a Simulink model.

The *MatlabEngineManager* class also handles the third part, *command evaluation*, sending over to the MATLAB instance the commands for execution. *Command result retrieval* from the evaluation of commands/functions works also as a factory. A `CommandResultRetriever`

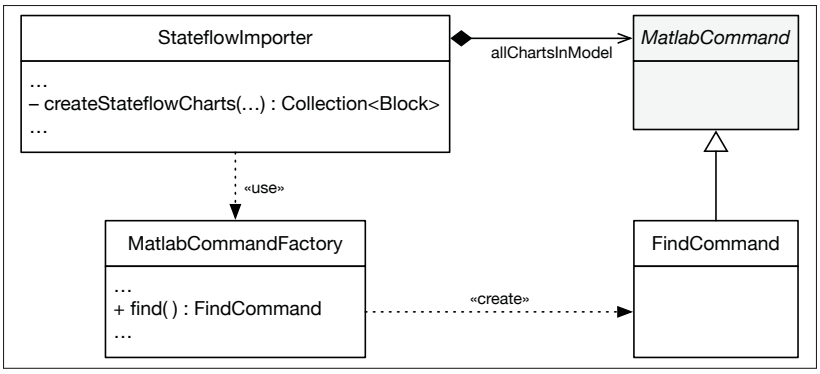


Figure-A V-5 Sample application of the factory design pattern.

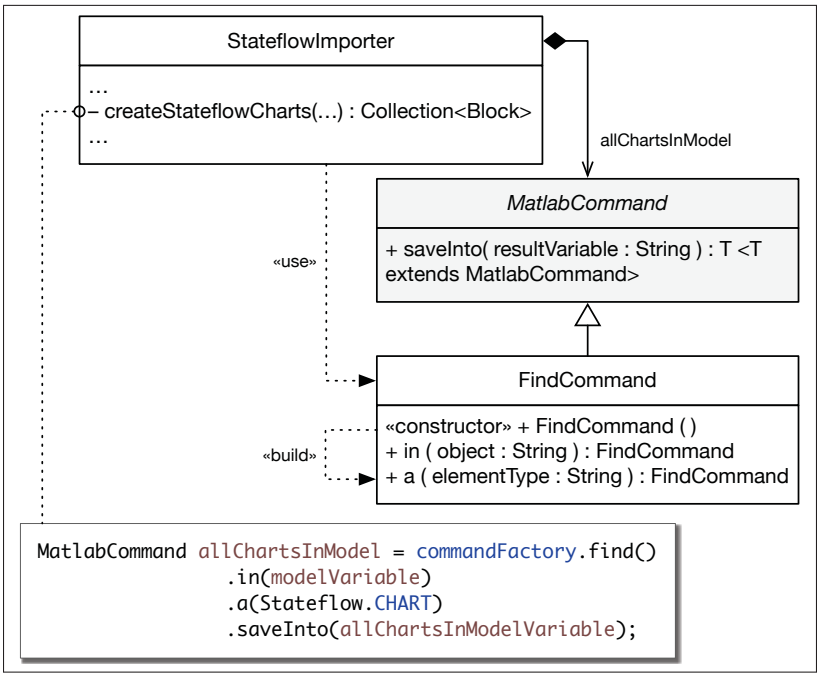


Figure-A V-6 Sample application of the builder design pattern.

class provides several factory methods that encapsulate long, repetitive result creation logic. These methods construct the required result object by extracting the necessary information from MATLAB's complex, string-based output.

5. Application in Avionic System Design

Breesse was motivated by the issues two industrial partners faced during their safety-critical avionics system developments. In essence: dealing with information that is spread across multiple models that are expressed in different modelling languages. The industrial partners represent avionics systems using a mix of UML, Simulink and Stateflow design models (Paz *et al.*, 2020). Manually ensuring consistency between such heterogeneous design models is a resource-consuming and error-prone activity for them. Hence, a *hybrid* approach was proposed (Paz *et al.*, 2020), where automated tools aid engineering teams in flagging errors for review and eventual correction. An Eclipse toolchain was implemented to support such approach. The toolchain is currently deployed in the industrial partners' premises. *Breesse* serves as a key constituent of the toolchain.

Various Simulink and Stateflow design models have been imported into Eclipse for their further processing with EMF-based technologies in the scope of the project: a Landing Gear System (LGS), a Flight Control System (FCS) and an Elevator Control System (ECS). Openly-available system descriptions of the LGS, FCS and ECS were developed by Paz & El Boussaidi (2018), Potter (2016) and Mosterman & Ghidella (2018), respectively. The Simulink and Stateflow design models for these three systems were created in close consultation with engineers of the industrial partners in order to achieve an accurate functional representation. Table-A V-1 provides a summary of their contents in terms of the number of model element types. Building an EMF-based representation is a computationally-demanding activity that depends on the design model's size. Table-A V-1 also presents the processing times to generate the EMF-based representations for the design models of the three avionics systems. Time measurements were taken on a Quad Core Intel Core i7 at 2.8 GHz with 16 GB of RAM. Figures-A V-7 and V-8 show excerpts of the design models for the LGS.

Figure-A V-9 displays a screenshot of V-7 and V-8. The left side shows the contextual menu option from which the import procedure can be invoked over a MATLAB `.slx` file. The right side shows an excerpt of the resulting EMF-based representation. The bottom of the right side

Table-A V-1 Summary of the Simulink and Stateflow design models for the LGS, FCS and ECS.

	LGS	**FCS**	**ECS**
Number of Simulink subsystem blocks	8	12	8
Number of Simulink inputs	7	31	8
Number of Simulink outputs	7	24	8
Number of Stateflow charts	3	3	6
Number of Stateflow states	23	6	19
Number of Stateflow transitions	31	6	30
Number of Stateflow junctions	0	0	2
Processing time	300 s	180 s	300 s

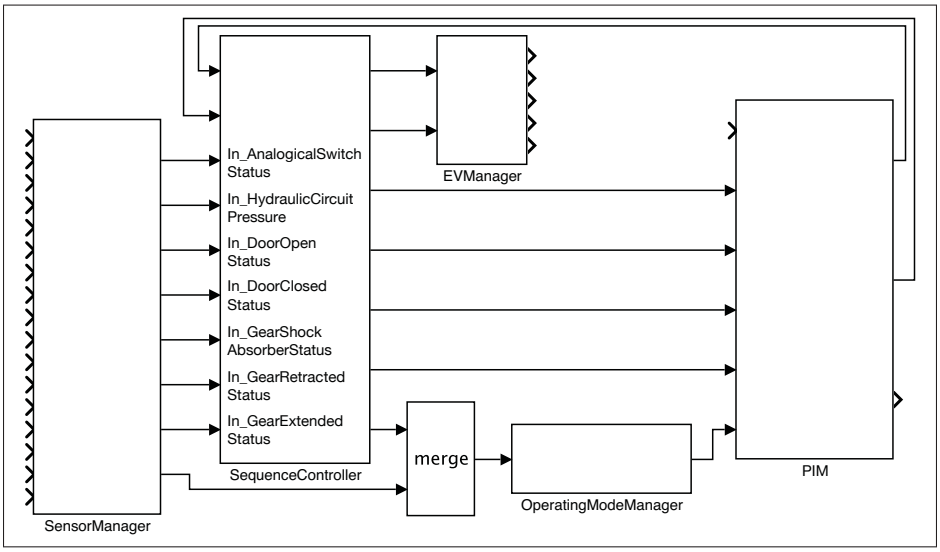


Figure-A V-7 Excerpt of the Simulink block diagram for the LGS.

shows an example of the properties in a Stateflow chart. Three professional engineers were recruited to verify the consistency between the original Simulink and Stateflow design models of the three systems and their resulting EMF-based representation. The engineers found all design models were correctly imported.

While *Breesse* has been developed and applied to representative avionics systems of the industrial partners, there is still much development to do. *Breesse*, especially its command evaluator framework, allows a precise data exchange with MATLAB through its MATLAB Engine API for Java, it is not possible to guarantee a flawless interaction. The MATLAB Engine API for

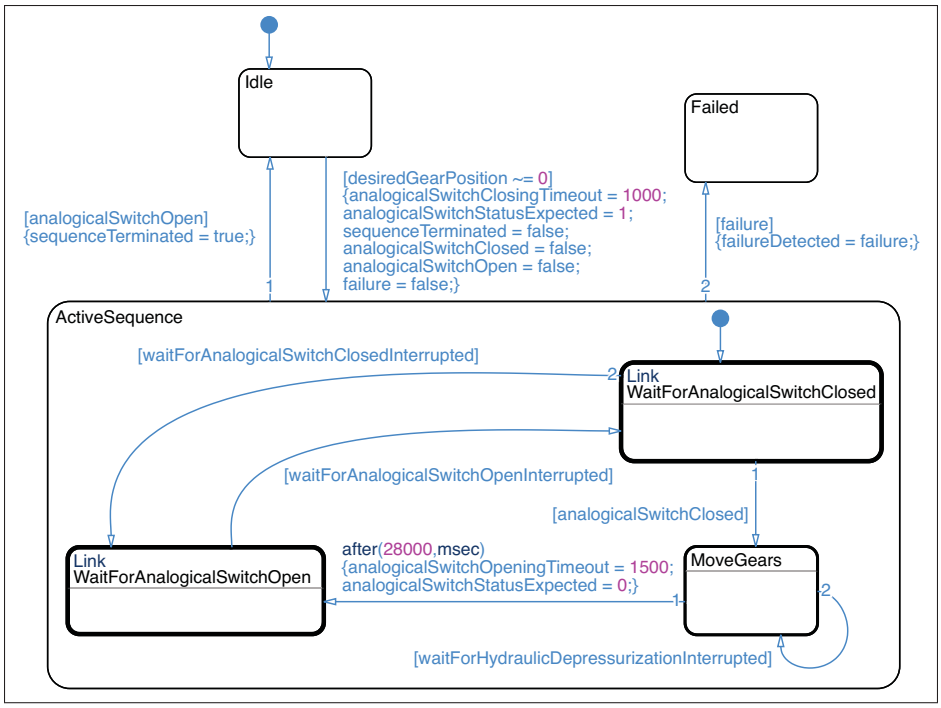


Figure-A V-8 Excerpt of a Stateflow chart for the LGS.

Java is limited in the size of the outputs returned from command/function evaluations. In the context of safety-critical embedded and cyber-physical systems this can be an issue when trying to satisfy design standards. Design standards specify methods, notations, rules, constraints, guidelines, and conventions to be used in the development of the design models. Certain elements in the resulting EMF-based representations from *Breesse* may end up with truncated data that violate established design standards. Scalability is also a major issue. MATLAB evaluates commands one-by-one. Depending on the amount and type of elements these evaluations may take longer for some commands. *Breesse* cannot obtain a result for a command until MATLAB has output a value. From the testing done with *Breesse*, design models with larger Stateflow structures (*e.g.*, the LGS and ECS design models) took a longer time to process than those with few Stateflow structures but larger Simulink structures (*e.g.*, FCS).

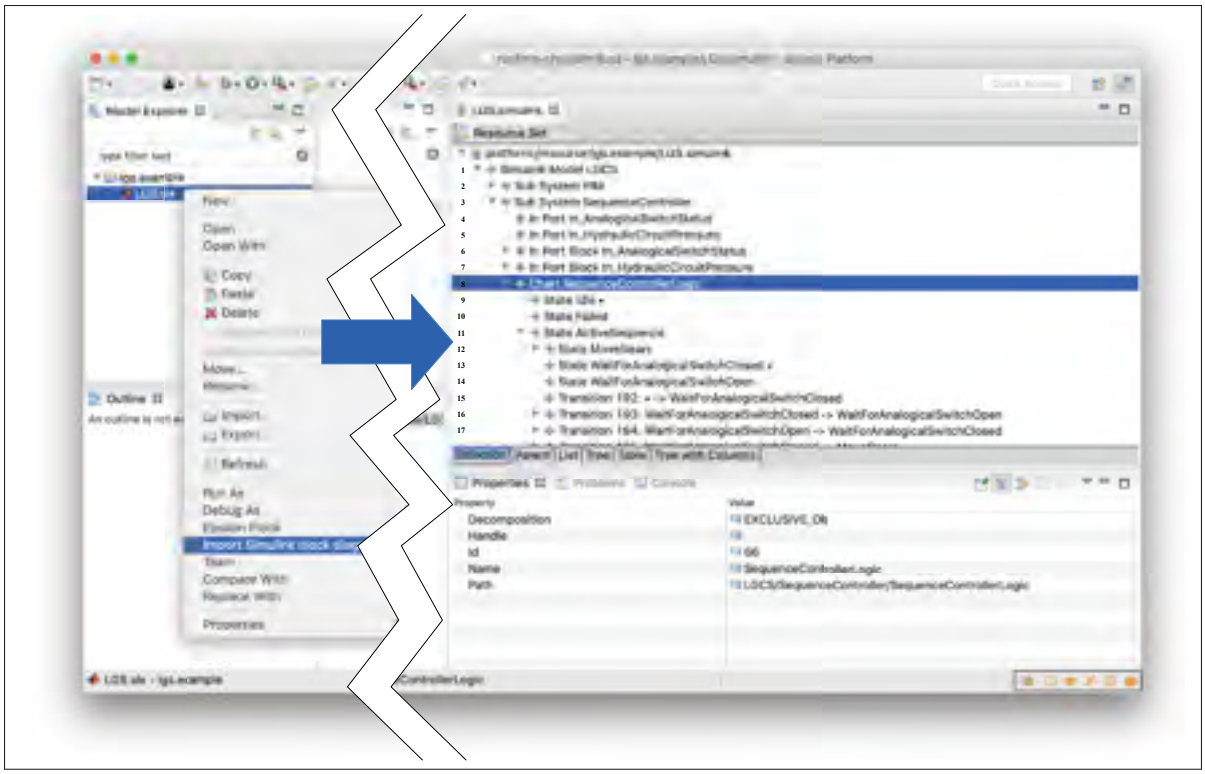


Figure-A V-9 Screenshot of the resulting EMF-based Simulink model for the LGS and the properties of the selected element in line 8.

APPENDIX VI

SPECML DOMAIN CONCEPTS

1. Constraint

The Constraint concept packages the expression of a constraint, which is defined over one or more parameters. Moreover, a constraint can be expressed as a conjunction of finer-grained constraints.

Attributes

- **expression** : *String [0..1]*

  Specifies the expression of the constraint.

Relationships

- **hasSubconstraints**

  Indicates the decomposition of the current constraint as a conjunction of subconstraints. The *subconstraints* end of the relationship indicates the current requirement is a conjunction of zero or more subconstraints. The *parent* end of the relationship indicates the constraint being decomposed.

- **overParameters**

  Indicates the parameters over which the constraint is defined. The constraint may involve one or more parameters.

2. `DataDictionary`

Description

A `DataDictionary` defines the container for the parameters used in the formalization constraints.

Relationships

- **hasEntries**

  Indicates the parameters considered as entries of the data dictionary. The *entries* end of the relationship indicates the entries in the dictionary. The *dataDictionary* end of the relationship indicates the data dictionary to which they belong.

3. `HighLevelRequirement`

Description

A `HighLevelRequirement` (HLR) specifies a capability or condition that must (or should) be satisfied. An HLR is developed from the analysis of SRATS, safety-related requirements and system architecture. HLRs must be very detailed so as to guide the design activities. HLRs must not include design nor verification details in accordance with DO-178C.

Generalizations

- `Requirement`

Attributes

- **precludesCFC** : *Boolean [1]*

  Indicates if the requirement intends to prevent one or more of the identified contributions to failure conditions (CFCs).

- **describesDesignDetail** : *Boolean [1]*

  Indicates if the requirement statement describes design detail. HLRs should not describe design details except when there is a justified design constraint.

- **describesVerificationDetail** : *Boolean [1]*

  Indicates if the requirement statement describes verification detail. HLRs should not describe verification details except when there is a justified constraint.

Relationships

- **copy**

  It could occur that allocated system requirements are, in fact, very detailed so as to guide the design without any further refinement into HLRs. In this case HLRs must be defined and related to their corresponding SRATS through the copy relationship. The Copy relationship goes from the HLR to the SRATS.

Constraints

1. A HighLevelRequirement without the isDerived flag must have a refinement or copy relationship to a SystemRequirement.

   This constraint is applicable to ensure traceability to an originating system requirement. Introduced from DO-178C.

2. A HighLevelRequirement with the describesDesignDetail flag must justify the existence of the design detail with a Rationale. Introduced from DO-178C.

   This constraint is applicable to enforce the inclusion of a rationale for DO-178C certification compliance.

3. A HighLevelRequirement with the describesVerificationDetail flag must justify the existence of the verification detail with a Rationale. Introduced from DO-178C.

   This constraint is applicable to enforce the inclusion of a rationale for DO-178C certification compliance.

4. LowLevelRequirement

Description

A LowLevelRequirement (LLR) specifies a capability or condition that must (or should) be satisfied. An LLR is developed from the HLRs and must be directly implementable/realizable without further information.

Generalizations

- Requirement

Constraints

1. A LowLevelRequirement without the isDerived flag must have a refinement relationship to a HighLevelRequirement.

   This constraint is applicable in adherence with DO-178C to ensure traceability to an originating HLR.

5. `Parameter`

Description

The `Parameter` concept represents an entry in the data dictionary. The concept defines the essential metadata.

Attributes

- **interpretation** : *String [1]*

  A description of the contents of the data and how it must be interpreted.

- **rationale** : *String [1]*

  A description of why the data is needed.

- **displayName** : *String [1]*

  The full readable name of the data.

- **source** : *Source [1..*]*

  The source who expresses the data. Possible values are: acquirer, operator, certification authority, specialty engineer, other stakeholder, or another source like certification standard, safety, costs, environmental conditions, design, production, tests, or maintenance. A data may have multiple sources.

- **creationNotes** : *String [*]*

  Notes about how the values for the data may be created.

- **maintenanceNotes** : *String [*]*

  Notes about how the values for the data may be obtained and/or updated.

- **usageNotes** : *String [*]*

  Information about the intended use of the data. May include one or more examples of values the data can take. Examples are intended to be illustrative.

- **creationDate** : *Date [1]*

  The date on which the data was created.

- **modificationDate** : *Date [1]*

  The date on which the data was last modified.

- **revision** : *Integer [1]*

  Indicates the number of times the data has been modified.

- **expectedRangeValues** : *String [1..*]*

  Indicates values for the data when the value is restricted, for instance to a list or a range of values.

- **defaultValue** : *String [1]*

  Indicates a default value for the data, within the `expectedRangeValues`, to be used on creation.

- **format** : *String [1]*

  Indicates the proper format for entering values to the data.

- **precision** : *Integer [1]*

  Indicates the precision allowed for the value of the data.

- **units** : *String [1]*

  Indicates the units used for the interpretation of the value of the data.

- **load** : *String [1]*

  Indicates the load at which the value of the data is updated.

- **isRequired** : *Boolean [1]*

  Indicates if the data is mandatory or optional.

- **isReadOnly** : *Boolean [1]*

  Indicates if the data is not meant to be changed.

- **isUnique** : *Boolean [1]*

  Indicates if the data i

- **encoding** : *String [1]*

  Indicates how the data is going to be stored for a later recall.

6. `PropertyBasedStatement`

Description

A `PropertyBasedStatement` establishes a requirement statement formalization following the PBR theory.

Attributes

- **id** : *String [1]*

  The unique ID of the statement.

- **text** : *String [1]*

  A textual description.

Relationships

- **hasCondition**

  Indicates the constraint representing the condition of actualization *C*. The condition is optional.

- **hasPredicate**

  Indicates the constraint representing the domain *D*, subset of $im(P)$, in which the value of a `Parameter` *P* of object *O* must be located when the condition *C* occurs or is achieved, *i.e.* $val(O.P) \in D \subset im(P)$.

- **specifies**

  Indicates the property-based statement specifies a behaviour that needs to be performed with a predetermined frequency for a number of times. The *timedEvent* end of the relationship indicates the annotation. The *when* end of the relationship indicates when the first occurrence occurs.

- **represents**

  Indicates the property-based statement represents an instant in time that will be observed for an event occurrence. The *timedInstantObservation* end of the relationship indicates the annotation. The *observedEvent* end of the relationship indicates when the associated observed event.

Constraints

1. The `condition` must not be a subconstraint.

   This constraint is applicable to avoid constraints that are not associated with a requirement.

2. The `predicate` must not be a subconstraint.

   This constraint is applicable to avoid constraints that are not associated with a requirement.

7. `Requirement`

Description

A `Requirement` defines the general attributes and relationships essential for requirements specification of avionics software in compliance with DO-178C. Specifically the concept generalizes the different kinds of requirements described in DO-178C, namely system requirements, high-level software requirements (HLRs) and low-level software requirements (LLRs). A `Requirement` conveys an understanding of what needs to be performed by the system of interest.

Attributes

- **id** : *String [1]*

  The unique ID of the requirement.

- **text** : *String [1]*

  The requirement statement as a natural language statement.

- **type** : *RequirementType [1]*

  The type of the requirement. Possible values are (Micouin, 2008): structural, behavioural or mixed. Structural requirements concern to a structural property (*i.e.* composition and structure) of the system being specified. Behavioural or functional requirements concern to a behavioural/functional property (*i.e.* observable behaviour) of the system being specified. Mixed requirements concern to both structural and behavioural properties of the system being specified.

- **source** : *Source [1..*]*

  The source who expresses the requirement. Possible values are: acquirer, operator, certification authority, specialty engineer, other stakeholder, or another source like certification

standard, safety, costs, environmental conditions, design, production, tests, or maintenance. A requirement may have multiple sources.

- **status** : *RequirementStatus [1]*

  The status of the reviewing and acceptance of the requirement. Possible values are: pending review, reviewed and accepted, and reviewed and incorrect.

- **conditionsType** : *ConditionsType [1]*

  The type of conditions expressed by the requirement. Possible values are: normal-range and abnormal-range (*i.e.* robustness). A normal-range requirement will describe the nominal behaviour of the software. On the other hand, a robustness requirement will describe the software's behaviour when its operating conditions are not normal.

- **isDerived** : *Boolean [1]*

  Indicates if a requirement is a derived requirement, *i.e.* a requirement that (a) is not directly traceable to a higher level requirement, and/or (b) specifies behaviour beyond that which is specified by the system requirements or the high-level requirements.

- **isStable** : *Boolean [1]*

  Indicates if a requirement is unlikely to be changed.

- **isVerifiable** : *Boolean [1]*

  Indicates if requirement-based verification activities can be carried out.

- **isConsistent** : *Boolean [1]*

  Indicates if the requirement is consistent according with the review of the requirement.

- **isFormalizable** : *Boolean [1]*

  Indicates if the requirement statement can be expressed using a formalism that can facilitate its analysis and verification.

- **revision** : *Integer [1]*

  Indicates the number of times the requirement has been modified.

- **creationDate** : *Date [1]*

  The date on which the requirement was created.

- **modificationDate** : *Date [1]*

  The date on which the requirement was last modified.

Relationships

- **refinement**

  Indicates the refinement of a requirement into lower level requirements. This relationship enables bi-directional traceability as required by DO-178C. The *refinedBy* end of the relationship indicates the current requirement is refined by zero or more lower level requirements. The *refines* end of the relationship indicates the higher level requirements being refined.

- **derivation**

  Indicates the derivation of a requirement. This relationship enables the indirect traceability between the derived requirement and an originating requirement. The *derivedBy* end of the relationship indicates the current requirement is derived by zero or more requirements. The *derives* end of the relationship indicates the requirement being refined.

- **coupling**

  Indicates the interdependence between requirements. This relationship enables the representation of interdependences that may exist between structural requirements and behavioural requirements. The *coupledTo* ends of the relationship indicate the coupled requirements. This relationship is included as a suggestion from the industrial partners.

- **justification**

  Indicates the rationale for the requirement's existence in the instances required by DO-178C (*e.g.*, when the requirement is a derived requirement).

- **formalizations**

  Indicates the formalizations of a requirement. The *formalizedBy* end of the relationship indicates the current requirement is formalized by zero or more property based statements. The *formalizes* end of the relationship indicates the requirement being formalized.

Constraints

1. The id must be specified and be unique.

   This constraint is applicable to uniquely identify a requirement.

2. A Requirement with the isDerived flag must be justified by a Rationale. Introduced from DO-178C.

   This constraint is applicable to understand why the requirement cannot be directly traced to an originating requirement because it specifies behaviour beyond that specified by the higher-level requirements.

8. SystemRequirement

Description

A SystemRequirement specifies a capability or condition that the system (or a part of it) must satisfy.

Generalizations

- Requirement

Attributes

- **isAllocatedToSoftware** : *Boolean [1]*

  Indicates if the system requirement has been allocated to software.

9. `TimedDurationConstraint`

Description

The `TimedDurationConstraint` concept imposes a constraint on the temporal distance between two behavioural events. This concept is included to allow the expression of timing constraints between specified behaviour in property-based statements.

Generalizations

- `Constraint`

Attributes

- **interpretation** : *String [1]*

  A description of the constraint and how it must be interpreted.

Relationships

- **from**

  Indicates the event that marks the start of the timing observation.

- **to**

  Indicates the event that marks the end of the timing observation.

10. `TimedEvent`

Description

The `TimedEvent` concept establishes an annotation on a property-based statement indicating that the specified behaviour needs to be performed with a predetermined frequency for a number of times.

Attributes

- **every** : *String [1]*

  Specifies the duration that separates successive occurrences of the timed event.

- **repetition** : *Integer [0..1]*

  Specifies the number of repetitive occurrences.

- **isRelative** : *Boolean [1]*

  Specifies whether the time value is considered to be relative (*i.e.* the when property is a time duration value) or absolute (*i.e.* the when property is a time instant value).

11. `TimedInstantObservation`

Description

The `TimedInstantObservation` concept denotes an instant in time that is associated with an event occurrence and observed. This concept is included to allow the observation of event occurrences and allowing their use in the expression of timing constraints on the specified behaviour.

Attributes

- **obsKind** : *EventKind [0..1]*

  Specifies the kind of the observed event. Possible values are: start, finish, send, receive, consume.

APPENDIX VII

SPECML STEREOTYPES

1. Copy

Description

The Copy relationship is as defined in the SysML specification but with a more constrained usage. A Copy relationship is a dependency between two requirements at different levels of abstraction, *e.g.*, a SystemRequirement and a HighLevelRequirement. The dependency specifies that the client requirement (*e.g.*, HLR) is a read-only copy of the supplier requirement (*e.g.*, an SRATS). The Copy relationship goes from the client requirement to the supplier requirement. For instance, this stereotype is included to make it possible to define SRATS as the HLRs when they are determined to be detailed enough to guide the design activities.

Generalizations

- SysML Copy

Attributes

(From UML Dependency class)

- **client** : *NamedElement [1..*]* (from)
- **supplier** : *NamedElement [1..*]* (to)

Constraints

1. Constraint 1 from the SysMLCopy stereotype is strengthened. A Copy dependency can only be created between subtypes of Requirement as follows:

- SystemRequirement (supplier) – HighLevelRequirement (client)
- HighLevelRequirement (supplier) – LowLevelRequirement (client)

2. Constraint 2 from the SysMLCopy stereotype. The text property of the client requirement is constrained to be a read-only copy of the supplier requirement.

3. Constraint 1 from the Trace stereotype. The Copy stereotype shall only be applied to dependencies.

2. Coupled

Description

Requirements at the same level of the requirements hierarchy may experience some interdependence. The Coupled stereotype makes it possible to represent such relationship between two requirements. For instance, this stereotype makes it possible to define the interdependence of a requirement of type StructuralRequirement with one of type BehaviouralRequirement. This stereotype is included as a suggestion from experienced industrial practitioners.

Generalizations

- SysML Trace

Attributes

(From UML Dependency class)

- **client** : *NamedElement [1..*]* (from)
- **supplier** : *NamedElement [1..*]* (to)

Constraints

1. Constraint 1 from the Trace stereotype. The Coupled stereotype shall only be applied to dependencies.

2. Constraint 2 from the Trace stereotype is strengthened. Dependencies stereotyped by Coupled shall have exactly one client and one supplier. Furthermore, dependencies with a Coupled stereotype applied must only relate two elements stereotyped by the same subtype of Requirement, one of which has type StructuralRequirement and the other one has type BehaviouralRequirement.

3. DataEntry

Description

The DataEntry stereotype represents an entry in the data dictionary. The stereotype defines the essential metadata.

Attributes

- **interpretation** : *String [1]*

  A description of the contents of the data and how it must be interpreted.

- **rationale** : *String [1]*

  A description of why the data is needed.

- **displayName** : *String [1]*

  The full readable name of the data.

- **source** : *Source [1..*]*

  The source who expresses the data. Possible values are: acquirer, operator, certification authority, specialty engineer, other stakeholder, or another source like certification standard, safety, costs, environmental conditions, design, production, tests, or maintenance. A data may have multiple sources.

- **creationNotes** : *String [*]*

  Notes about how the values for the data may be created.

- **maintenanceNotes** : *String [*]*

  Notes about how the values for the data may be obtained and/or updated.

- **usageNotes** : *String [*]*

  Information about the intended use of the data. May include one or more examples of values the data can take. Examples are intended to be illustrative.

- **creationDate** : *Date [1]*

  The date on which the data was created.

- **modificationDate** : *Date [1]*

  The date on which the data was last modified.

- **revision** : *Integer [1]*

  Indicates the number of times the data has been modified.

- **expectedRangeValues** : *String [1..*]*

  Indicates values for the data when the value is restricted, for instance to a list or a range.

- **defaultValue** : *String [1]*

  Indicates a default value for the data, within the `expectedRangeValues`, to be used on creation.

- **format** : *String [1]*

  Indicates the proper format for entering values to the data.

- **precision** : *Integer [1]*

  Indicates the precision allowed for the value of the data.

- **units** : *String [1]*

  Indicates the units used for the interpretation of the value of the data.

- **load** : *String [1]*

  Indicates the load at which the value of the data is updated.

- **isRequired** : *Boolean [1]*

  Indicates if the data is mandatory or optional.

- **isReadOnly** : *Boolean [1]*

  Indicates if the data is not meant to be changed.

- **isUnique** : *Boolean [1]*

  Indicates if the data i

- **encoding** : *String [1]*

  Indicates how the data is going to be stored for a later recall.

4. Derive

Description

A Derive relationship is a dependency between two requirements, a subtype of Requirement with the isDerived flag and another subtype of Requirement without the isDerived flag. The dependency specifies that the client requirement (*i.e.* the requirement with the isDerived flag) depends on the supplier requirement (*i.e.* the requirement without the isDerived flag), which

is the requirement from which the former was derived. This stereotype is included to make it possible to trace the derived requirement indirectly to a higher level requirement through the mediation of the requirement from which it was derived.

Generalizations

- SysML Trace

Attributes

(From UML Dependency class)

- **client** : *NamedElement [1..*]* (from)
- **supplier** : *NamedElement [1..*]* (to)

Constraints

1. Constraint 1 from the `Trace` stereotype. The `Derive` stereotype shall only be applied to dependencies.
2. Constraint 2 from the `Trace` stereotype is strengthened. Dependencies stereotyped by `Derive` shall have exactly one client and one supplier. Furthermore, a `Derive` dependency can only be created between a subclass of `Requirement` with the `isDerived` attribute set to `true` (client) and a subclass of `Requirement` with the `isDerived` attribute set to `false` (client).

5. `HighLevelRequirement`

Description

A `HighLevelRequirement` (HLR) specifies a capability or condition that must (or should) be satisfied. An HLR is developed from the analysis of SRATS, safety-related requirements and system architecture. HLRs must be very detailed so as to guide the design activities. It could occur that allocated system requirements are, in fact, very detailed so as to guide the design without any further refinement into HLRs. In this case HLRs must be defined and related to their corresponding SRATS using the `Copy` relationship. The `Copy` relationship goes from the HLR to the SRATS. HLRs must not include design details nor include verification details in accordance with regulations.

Generalizations

- `Requirement`

Attributes

- **precludesCFC** : *Boolean [1]*

  Indicates if the requirement intends to prevent one or more of the identified contributions to failure conditions (CFCs).

- **describesDesignDetail** : *Boolean [1]*

  Indicates if the requirement statement describes design detail. HLRs should not describe design details except when there is a justified design constraint.

- **describesVerificationDetail** : *Boolean [1]*

  Indicates if the requirement statement describes verification detail. HLRs should not describe verification details except when there is a justified constraint.

Constraints

1. A HighLevelRequirement stereotype must not be applied alongside other stereotypes that specialize the Requirement stereotype.

2. A HighLevelRequirement without the isDerived flag must have a RefineReqt or Copy dependency to a SystemRequirement.

   This constraint is applicable to ensure traceability to an originating system requirement.

3. A HighLevelRequirement with the describesDesignDetail flag must justify the existence of the design detail with a Rationale (see clause 7.3.2.5 Rationale from the OMG (2017a) SysML 1.5 Specification).

   This constraint is applicable to enforce the inclusion of a rationale for certification compliance.

4. A HighLevelRequirement with the describesVerificationDetail flag must justify the existence of the verification detail with a Rationale (see clause 7.3.2.5 Rationale from the OMG (2017a) SysML 1.5 Specification).

   This constraint is applicable to enforce the inclusion of a rationale for certification compliance.

6. LowLevelRequirement

Description

A LowLevelRequirement (LLR) specifies a capability or condition that must (or should) be satisfied. An LLR is developed from the HLRs and must be directly implementable/realizable without further information. The stereotype can be used: 1) as a standalone element to capture natural language or formal requirement statements, or 2) to stereotype a design model element as an LLR.

Generalizations

- Requirement

Constraints

1. A LowLevelRequirement stereotype must not be applied alongside other stereotypes that specialize the Requirement stereotype.

2. A LowLevelRequirement without the isDerived flag must have a RefineReqt dependency to a HighLevelRequirement.

   This constraint is applicable in adherence with DO-178C to ensure traceability to an originating HLR.

7. PropertyBasedStatement

Description

A PropertyBasedStatement establishes a requirement statement formalization following the PBR theory.

Extensions

- UML Class

Attributes

- **id** : *String [1]*

  The unique ID of the statement.

- **text** : *String [1]*

  A textual description.

- **condition** : *ConstraintBlock [0..1]*

  The condition of actualization *C*. The condition is represented as a ConstraintBlock. The condition is optional.

- **predicate** : *ConstraintBlock [1]*

  The domain *D* subset of $im(P)$ in which the value of property *P* of object *O* must be located when the condition *C* occurs or is achieved, *i.e.* $val(O.P) \in D \subset im(P)$. The predicate is represented as a ConstraintBlock.

Constraints

1. A PropertyBasedStatement shall not own any structural or behavioural elements beyond the properties that define its condition and predicate expressions.
2. A PropertyBasedStatement shall not participate in associations other than the one that binds it to the Requirement that it formalizes.
3. A PropertyBasedStatement shall not participate in generalizations.
4. A PropertyBasedStatement shall not have nested classifiers.
5. A PropertyBasedStatement stereotype must not be applied alongside other stereotypes.

8. RefineReqt

Description

A RefineReqt relationship is a bidirectional trace in which a requirement can be developed into a lower-level requirement. This stereotype is included to trace system requirements that

are developed into HLRs and, in turn, HLRs are developed into LLRs. The `RefineReqt` relationship goes from the refining requirement (*e.g.*, the LLR) to the refined requirement (*e.g.*, the HLR).

Generalizations

- SysML DeriveReqt

Attributes

(From UML Dependency class)

- **client** : *NamedElement [1..*]* (from)
- **supplier** : *NamedElement [1..*]* (to)

Constraints

1. Constraint 1 from the `Trace` stereotype. The `RefineReqt` stereotype shall only be applied to dependencies.
2. The refined requirement (supplier) shall be an element stereotyped by a subtype of `Requirement`.
3. The refining requirement (client) shall be an element stereotyped by a subtype of `Requirement`.
4. Constraint 2 from the `Trace` stereotype is strengthened. Dependencies with a `RefineReqt` stereotype applied shall have exactly one refined requirement and one refining requirement as follows:
   - SystemRequirement (refined) – HighLevelRequirement (refining)
   - HighLevelRequirement (refined) – LowLevelRequirement (refining)

9. Requirement

Description

A Requirement defines the general attributes and relationships essential for requirements specification of avionics software. Specifically the stereotype generalizes the different kinds of requirements described in DO-178C, namely system requirements, high-level software requirements (HLRs) and low-level software requirements (LLRs). A Requirement conveys an understanding of what needs to be performed by the system of interest.

Extensions

- UML Class

Generalizations

- SysML AbstractRequirement

Attributes

- **id** : *String [1]*

  The unique ID of the requirement. Inherited from AbstractRequirement (see clause 16.3.2.1 AbstractRequirement from the OMG (2017a) SysML 1.5 Specification).

- **text** : *String [1]*

  The requirement statement as a natural language statement. Inherited from AbstractRequirement (see clause 16.3.2.1 AbstractRequirement from the OMG (2017a) SysML 1.5 Specification).

- **type** : *RequirementType [1]*

  The type of the requirement. Possible values are (Micouin, 2008): structural, behavioural or mixed. Structural requirements concern to a structural property (*i.e.* composition and structure) of the system being specified. Behavioural or functional requirements concern to a behavioural/functional property (*i.e.* observable behaviour) of the system being specified. Mixed requirements concern to both structural and behavioural properties of the system being specified.

- **source** : *Source [1..*]*

  The source who expresses the requirement. Possible values are: acquirer, operator, certification authority, specialty engineer, other stakeholder, or another source like certification standard, safety, costs, environmental conditions, design, production, tests, or maintenance. A requirement may have multiple sources.

- **status** : *RequirementStatus [1]*

  The status of the reviewing and acceptance of the requirement. Possible values are: pending review, reviewed and accepted, and reviewed and incorrect.

- **conditionsType** : *ConditionsType [1]*

  The type of conditions expressed by the requirement. Possible values are: normal-range and abnormal-range (*i.e.* robustness). A normal-range requirement will describe the nominal behaviour of the software. On the other hand, a robustness requirement will describe the software's behaviour when its operating conditions are not normal.

- **isDerived** : *Boolean [1]*

  Indicates if a requirement is a derived requirement, *i.e.* a requirement that (a) is not directly traceable to a higher level requirement, and/or (b) specifies behaviour beyond that which is specified by the system requirements or the high-level requirements.

- **isStable** : *Boolean [1]*

  Indicates if a requirement is unlikely to be changed.

- **isVerifiable** : *Boolean [1]*

  Indicates if requirement-based verification activities can be carried out.

- **isConsistent** : *Boolean [1]*

  Indicates if the requirement is consistent according with the review of the requirement.

- **isFormalizable** : *Boolean [1]*

  Indicates if the requirement statement can be expressed using a formalism that can facilitate its analysis and verification.

- **revision** : *Integer [1]*

  Indicates the number of times the requirement has been modified.

- **creationDate** : *Date [1]*

  The date on which the requirement was created.

- **modificationDate** : *Date [1]*

  The date on which the requirement was last modified.

- **/derived** : *AbstractRequirement [*]*

  Inherited from `AbstractRequirement` (see clause 16.3.2.1 AbstractRequirement from the OMG (2017a) SysML 1.5 Specification).

- **/derivedFrom** : *AbstractRequirement [*]*

  Inherited from `AbstractRequirement` (see clause 16.3.2.1 AbstractRequirement from the OMG (2017a) SysML 1.5 Specification).

- **/satisfiedBy** : *NamedElement [*]*

  Inherited from `AbstractRequirement` (see clause 16.3.2.1 AbstractRequirement from the OMG (2017a) SysML 1.5 Specification).

- **/tracedTo** : *NamedElement [*]*

  Inherited from AbstractRequirement (see clause 16.3.2.1 AbstractRequirement from the OMG (2017a) SysML 1.5 Specification).

- **/verifiedBy** : *NamedElement [*]*

  Inherited from AbstractRequirement (see clause 16.3.2.1 AbstractRequirement from the OMG (2017a) SysML 1.5 Specification).

- **/master** : *AbstractRequirement*

  Inherited from AbstractRequirement (see clause 16.3.2.1 AbstractRequirement from the OMG (2017a) SysML 1.5 Specification).

- **/formalization** : *PropertyBasedStatement [*]*

  Indicates the requirement's formalization as a collection of property-based statements. One property-based statement may not be sufficient to describe the entire domain captured in the requirement's text attribute. Thus, a collection of property-based statements may be introduced as part of the requirement's formalization. The requirement is, therefore, interpreted as the conjunction of the specified property-based statements.

Constraints

1. The id must be specified and be unique.

   This constraint is applicable to uniquely identify a requirement.

2. A Requirement with the isDerived flag must be justified by a Rationale (see clause 7.3.2.5 Rationale from the OMG (2017a) SysML 1.5 Specification).

   This constraint is applicable to understand why the requirement cannot be directly traced to an originating requirement because it specifies behaviour beyond that specified by the higher-level requirements.

3. A Requirement, or any of its subclasses, shall not participate as the client in dependencies stereotyped by Satisfy.

   The Satisfy relationship is a dependency between a requirement and a model element that fulfills the requirement. However, when the model element is stereotyped as a LowLevelRequirement the Satisfy relationship takes on the same meaning as the RefineReqt relationship. In such case the RefineReqt relationship shall be used. Thus, this constraint is intended to enforce the use of the RefineReqt relationship over the Satisfy relationship.

4. A Requirement, or any of its subclasses, shall not own any nested classifiers stereotyped by Requirement.

   This constraint is to avoid the creation of compound requirements and subrequirements.

10. SystemRequirement

Description

A SystemRequirement specifies a capability or condition that the system (or a part of it) must satisfy.

Generalizations

- Requirement

Attributes

- **isAllocatedToSoftware** : *Boolean [1]*

  Indicates if the system requirement has been allocated to software.

Constraints

1. A SystemRequirement stereotype must not be applied alongside other stereotypes that specialize the Requirement stereotype.

11. TimedDomain

Description

The TimedDomain stereotype is as defined by the MARTE specification. A package stereotyped by TimedDomain is a container of clocks. Elements in a TimedDomain package may use the clocks contained in it to express behaviour that is time dependent. A TimedDomain package may own nested TimedDomain packages.

Generalizations

- MARTE TimedDomain

12. TimedDurationConstraint

Description

The TimedDurationConstraint stereotype is as defined by the MARTE specification. The stereotype imposes a constraint on the temporal distance between two events. This stereotype is included to allow the expression of timing constraints between specified behaviour.

Generalizations

- MARTE TimedDurationConstraint

Attributes

- **kind** : *MARTE::NFP::NFP_Annotation::ConstraintKind [0..1]*

  Specifies the kind of the constraint. The constraint can be qualified by either a *required*, *offered* or *contract* kind. A constraint that is *required* indicates the minimum level that is demanded for the model element. A constraint that is *offered* establishes the space of values that can support the model element. A constraint that is *contract* defines a conditional expression between offered and required values.

- **interpretation** : *TimeInterpretationKind [1]*

  The value of this attribute is fixed to specify that the constraint applies to a duration value.

- **on** : *MARTE::Time::TimeAccesses::Clocks::Clock [1..*]*

  Specifies the associated clock(s).

- **specification** : *DurationPredicate [1]*

  Specifies the constrained duration value.

Constraints

1. The owner of an element stereotyped by `TimedDurationConstraint` must be a package stereotyped by `TimedDomain`.

13. `TimedEvent`

Description

The `TimedEvent` stereotype is as defined by the MARTE specification but with a more constrained usage. The stereotype establishes a non-functional annotation on a `PropertyBasedStatement` indicating the specified behaviour needs to be performed with a predetermined frequency (*i.e.* it is explicitly bound to a clock).

Generalizations

- MARTE TimedEvent

Attributes

- **when** : *CVS::ClockedValueSpecification [1]*

  Specifies when the first occurrence occurs.

- **every** : *CVS::DurationValueSpecification [0..1]*

  Specifies the duration that separates successive occurrences of the timed event.

- **repetition** : *Integer [0..1]*

  Specifies the number of repetitive occurrences.

- **isRelative** : *ConstraintBlock [1]*

  Specifies whether the time value is relative (*i.e.* the when property is a time duration value) or absolute (*i.e.* the when property is a time instant value).

Constraints

1. A `TimedEvent` stereotype must only be applied to an element stereotyped by `PropertyBasedStatement`.

2. The owner of an element stereotyped by `TimedEvent` must be a package stereotyped by `TimedDomain`.

14. TimedInstantObservation

Description

The TimedInstantObservation stereotype is as defined by the MARTE specification but with a more constrained usage. The stereotype denotes an instant in time that is associated with an event occurrence and observed on a given clock. This stereotype is included to allow the observation of event occurrences and allowing their use in the expression of timing constraints on the specified behaviour. The stereotype must only be applied to a property-based statement.

Generalizations

- MARTE TimedInstantObservation

Attributes

- **obsKind** : *EventKind [0..1]*

  Specifies the kind of the observed event. Possible values are: start, finish, send, receive, consume.

- **eocc** : *MARTE::CoreElements::Causality::RunTimeContext::EventOccurrence [1]*

  The associated observed event.

- **on** : *MARTE::Time::TimeAccesses::Clocks::Clock [1..*]*

  Specifies the associated clock(s).

Constraints

1. A TimedInstantObservation stereotype must only be applied to a PropertyBased-Statement.

2. The owner of an element stereotyped by TimedInstantObservation must be a package stereotyped by TimedDomain.

APPENDIX VIII

SPECML DEVELOPER'S AND USER'S GUIDES

1. Developer's Guide

1.1 Getting started

System requirements

- Java JDK 8 or higher
- Eclipse Oxygen or newer with the Modeling package (*a.k.a.* Eclipse Modeling Tools)

Install required dependencies

The required dependency is Papyrus modelling environment.

1. Go to **Help** » **Install New Software...**.
2. Work with http://download.eclipse.org/modeling/mdt/papyrus/updates/releases/oxygen to install the Papyrus modelling environment. Select the following features:
   - Papyrus
   - Papyrus Toolsmiths

Install Papyrus additional components

1. Go to **Help** » **Install Papyrus Additional Components**.
2. Select:
   - Papyrus DSML Validation
   - Papyrus SysML 1.4

Install SpecML

1. Due to a bug in Papyrus, you must place the SpecML plug-ins under `«path to Eclipse »/dropins`. **Note:** These plug-ins must be updated with every change to the source code.
2. (Re)Start the Eclipse Papyrus modelling environment.

Import SpecML projects

1. Import the SpecML source code projects into the Eclipse workspace.

Create new run configuration

Create a new Run Configuration in Eclipse:

1. Go to **Run** » **Run Configurations...**.
2. From the list in the left, double click on **Eclipse Application**.
3. Give the configuration a name (suggestion: **SpecML**).
4. In the **Plug-ins** tab, select to launch with **plug-ins selected below only**.
5. Under **Workspace**, select the plug-ins that correspond to SpecML.
6. Uncheck the box **Add new workspace Plug-ins to this launch configuration automatically**.
7. Uncheck the box **Validate Plug-ins automatically prior to launching**.
8. Click on **Apply**.

1.2 Project structure

- **src-gen**

  This folder contains generated source code from models. These files should not be edited directly.

- **src**

  This folder contains manually created source code. These files can be edited directly if necessary.

- **icons**

  This folder contains image files used for the icons in SpecML's reference implementation.

- **META-INF** and **MANIFEST.MF**

  This folder and file contain descriptions of the extensions to the Eclipse Papyrus modelling environment provided by the plug-in.

- **models**

  This folder contains EMF models with extensions to the Eclipse Papyrus modelling environment.

- **resources**

  This folder contains other resources used in the extensions to the Eclipse Papyrus modelling environment.

- **build.properties**

  This file contains the build properties of the plug-in.

- **plugin.properties**

  This file contains additional properties of the plug-in.

- **plugin.xml**

  This file contains descriptions of the extensions to the Eclipse Papyrus modelling environment provided by the plug-in.

1.3 SpecML projects and noteworthy files

- **ca.ets.sofeess.specml.architecture**

  This project specifies a new *Avionics* architecture context and the *SpecML* viewpoint in the Eclipse Papyrus modelling environment.

  – **specml.architecture**

    EMF model specifying the *Avionics* architecture context and *SpecML* viewpoint. Basic concepts on architecture models and a walkthrough for the definition of new architecture models can be found at https://help.eclipse.org/oxygen/topic/org.eclipse.papyrus.infra.architecture.doc/target/generated-eclipse-help/architecture.html?cp=79_1_3.

  – **SpecModelCreationCommand.java**

    Initializes a new SpecML model by applying the SpecML profile.

- **ca.ets.sofeess.specml.newchild**

  This project specifies the necessary facilities for the creation and manipulation of SpecML constructs within UML models.

  – **newChild.creationmenumodel**

    EMF model specifying a context menu for the creation of SpecML constructs within a UML model. Basic concepts and how to create or modify creation menu models can be found at https://help.eclipse.org/oxygen/topic/org.eclipse.papyrus.infra.newchild.doc/target/generated-eclipse-help/newChild.html?cp=79_1_6.

  - **specml.elementtypesconfigurations**

    EMF model specifying the high-level model editing facilities for UML models necessary to support the creation and manipulation of SpecML constructs through the context menu. Basic concepts on the ElementTypeConfigurations Framework can be found at https://help.eclipse.org/oxygen/topic/org.eclipse.papyrus.infra.types.doc/target/generated-eclipse-help/types.html?cp=79_1_7.

- **ca.ets.sofeess.specml.palette**

  This project provides a palette in the main SpecML model editor for quick access to SpecML constructs. The palette is organized into categories for a better reference.

  - **specml.paletteconfiguration**

    EMF model specifying the elements in the palette and their categories. The Eclipse Papyrus modelling environment provides creation forms for the contents of this model. A demonstration can be found at https://www.youtube.com/watch?v=XnhxHPksbjc.

  - **specmldi.elementtypesconfigurations**

    EMF model specifying the high-level model editing facilities for UML models necessary to support the creation and manipulation of SpecML constructs through the palette. Basic concepts on the ElementTypeConfigurations Framework can be found at https://help.eclipse.org/oxygen/topic/org.eclipse.papyrus.infra.types.doc/target/generated-eclipse-help/types.html?cp=79_1_7.

- **ca.ets.sofeess.specml.profile**

  This project specifies the SpecML profile.

  - **specml.profile**

    Papyrus Profile resource model file defining the SpecML profile, its stereotypes and OCL constraints. Java code must be generated from this model through an EMF generator model (genmodel). Basic concepts and a walkthrough on defining profiles and

stereotypes in the Eclipse Papyrus modelling environment can be found at https://help.eclipse.org/oxygen/nav/79_0_1_6.

- **ca.ets.sofeess.specml.properties**

  This project provides an extension to the *Properties* view in the Eclipse Papyrus modelling environment for editing properties of the SpecML profile that are too hard to edit directly from the main graphical editor or that do not have a graphical representation.

  – **specml.ctx**

    EMF model specifying the properties from the SpecML stereotypes that can be edited, the types of widgets to enable editing, and their organization in a form-like view. Basic concepts and a walkthrough on the Properties View framework can be found at https://help.eclipse.org/oxygen/topic/org.eclipse.papyrus.views.properties.doc/target/generated-eclipse-help/properties-view.html?cp=79_1_0.

- **ca.ets.sofeess.specml.style**

  This project specifies custom graphical representations for certain SpecML stereotypes. This is done through the use of Cascading StyleSheets (CSS). Basic concepts and a walkthrough on graphical customizations with CSS can be found at https://help.eclipse.org/oxygen/topic/org.eclipse.papyrus.infra.gmfdiag.css.doc/target/generated-eclipse-help/css.html?cp=79_1_5.

- **ca.ets.sofeess.specml.tables**

  This project provides tabular representations for certain SpecML constructs. Tabular representations are available for SRATS, HLRs and property-based statements. These tabular representations can be modified or new ones can be added. Basic concepts on tables and a walkthrough for table creation can be found at https://help.eclipse.org/oxygen/nav/79_2_4.

- **ca.ets.sofeess.specml.validation**

  This project provides constraint validation facilities. This project is generated from the *SpecML.profile* file in the *ca.ets.sofeess.specml.profile* project. Basic concepts on con-

straint validation and a walkthrough on generating the validation plug-in project can be found at https://help.eclipse.org/oxygen/topic/org.eclipse.papyrus.dsml.validation.doc/target/generated-eclipse-help/dsml-validation.html?cp=79_1_2.

1.4 More information

More information about the Eclipse Papyrus modelling environment and how to customize it for a particular domain is available at https://help.eclipse.org/oxygen/nav/79_1 and http://wiki.eclipse.org/Papyrus_Developer_Guide?cp=79_2. A YouTube video with more explanations and live demonstrations of the Eclipse Papyrus modelling environment is available at https://www.youtube.com/watch?v=U62b2EQObRg.

1.5 Limitations

SpecML's reference implementation is limited by the Eclipse Papyrus modelling environment. Some of the features of SpecML could not be properly implemented because of limitations in Papyrus' UML, SysML and MARTE implementations. Papyrus does not implement SysML 1.5, only SysML 1.4, which limits extensions that can be done to the SysML`Requirement` stereotype. Hence, the `AbstractRequirement` stereotype of SysML 1.5 was manually defined for SpecML's reference implementation. Papyrus' implementation of the MARTE profile was archived and could not be used successfully to integrate it with Papyrus' current implementation of UML. Therefore, stereotypes that were borrowed from MARTE had to be manually defined for SpecML's reference implementation. The previous implementation choices by no means limit SpecML as its reference implementation can be developed differently.

2. User's Guide

2.1 Getting started

System requirements

- Java JDK 8 or newer
- Eclipse Oxygen or newer with the Modeling package (*a.k.a.* Eclipse Modeling Tools)

Install required dependencies

The required dependency is Papyrus modelling environment.

1. Go to **Help** » **Install New Software...**.
2. Work with http://download.eclipse.org/modeling/mdt/papyrus/updates/releases/oxygen to install the Papyrus modelling environment. Select the following features:
   - Papyrus

Install Papyrus additional components

1. Go to **Help** » **Install Papyrus Additional Components**.
2. Select:
   - Papyrus DSML Validation
   - Papyrus SysML 1.4

More information about the Eclipse Papyrus modelling environment and how to use it is available at https://help.eclipse.org/oxygen/nav/79_0.

Install SpecML

1. Place the SpecML plug-ins under «path to Eclipse »/plugins.

2. (Re)Start the Eclipse Papyrus modelling environment.

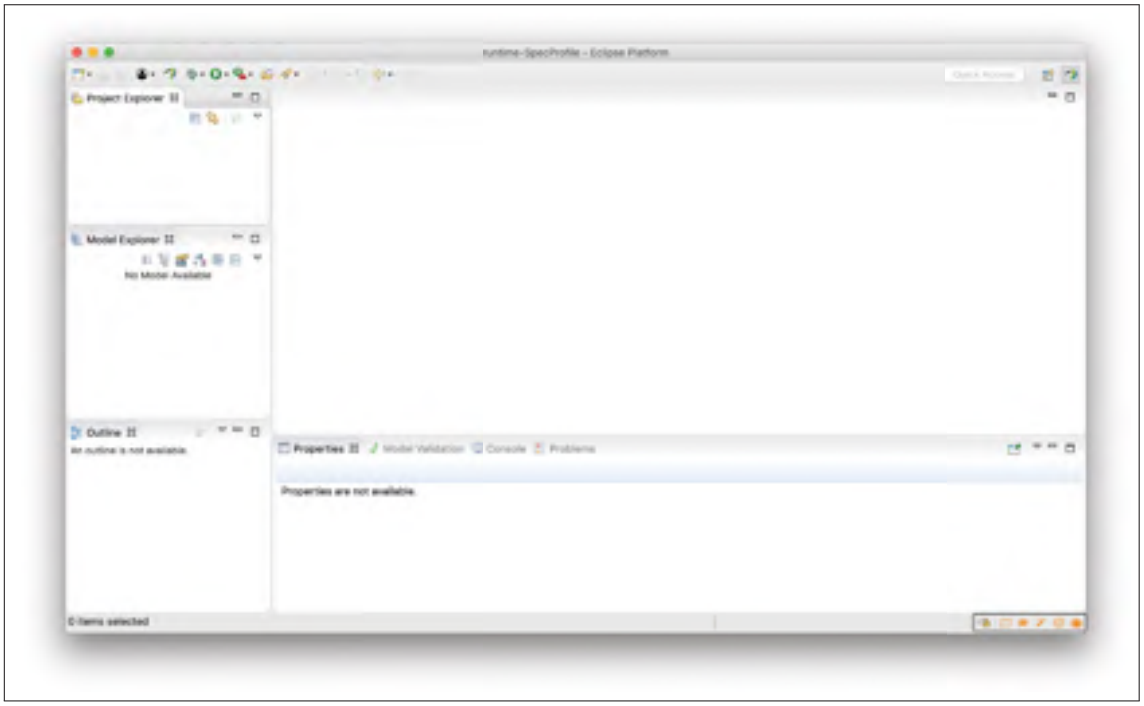
SpecML plug-ins

- ca.ets.sofeess.specml.architecture_1.0.0.jar
- ca.ets.sofeess.specml.newchild_1.0.0.jar
- ca.ets.sofeess.specml.palette_1.0.0.jar
- ca.ets.sofeess.specml.profile_1.0.0.jar
- ca.ets.sofeess.specml.properties_1.0.0.jar
- ca.ets.sofeess.specml.style_1.0.0.jar
- ca.ets.sofeess.specml.tables_1.0.0.jar
- ca.ets.sofeess.specml.validation_1.0.0.jar

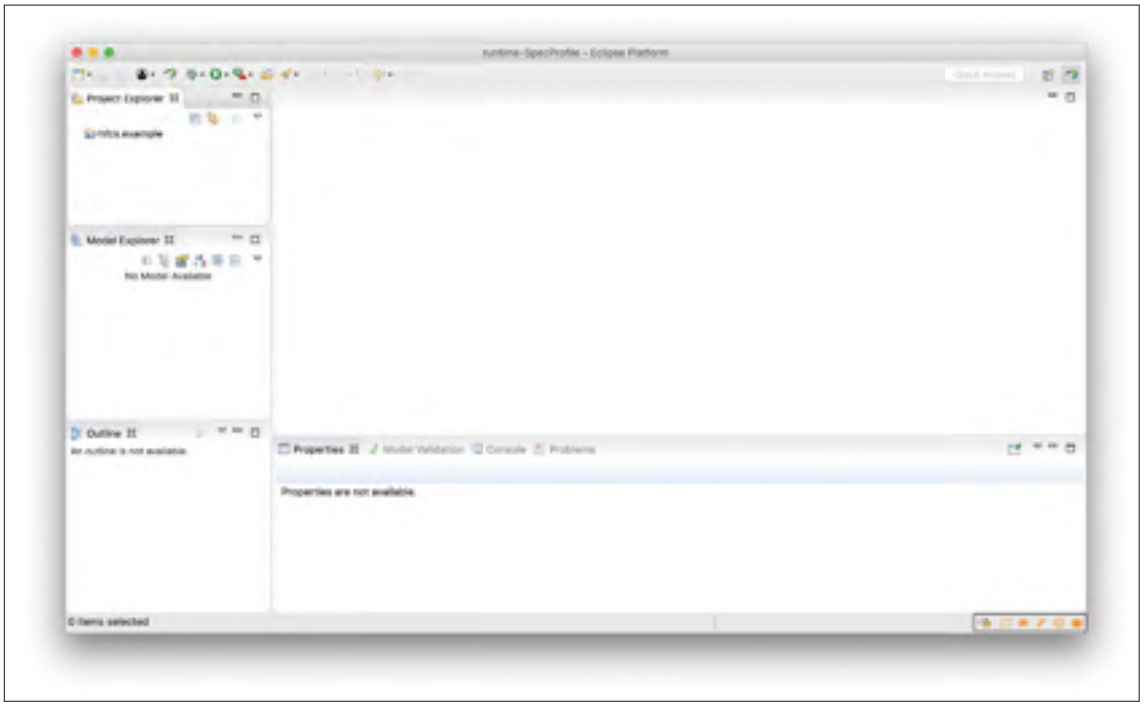
2.2 Creating a specification model

The following steps guide the creation of a new SpecML specification model, its verification of well-formedness, and the creation of tabular views of the requirements specification.

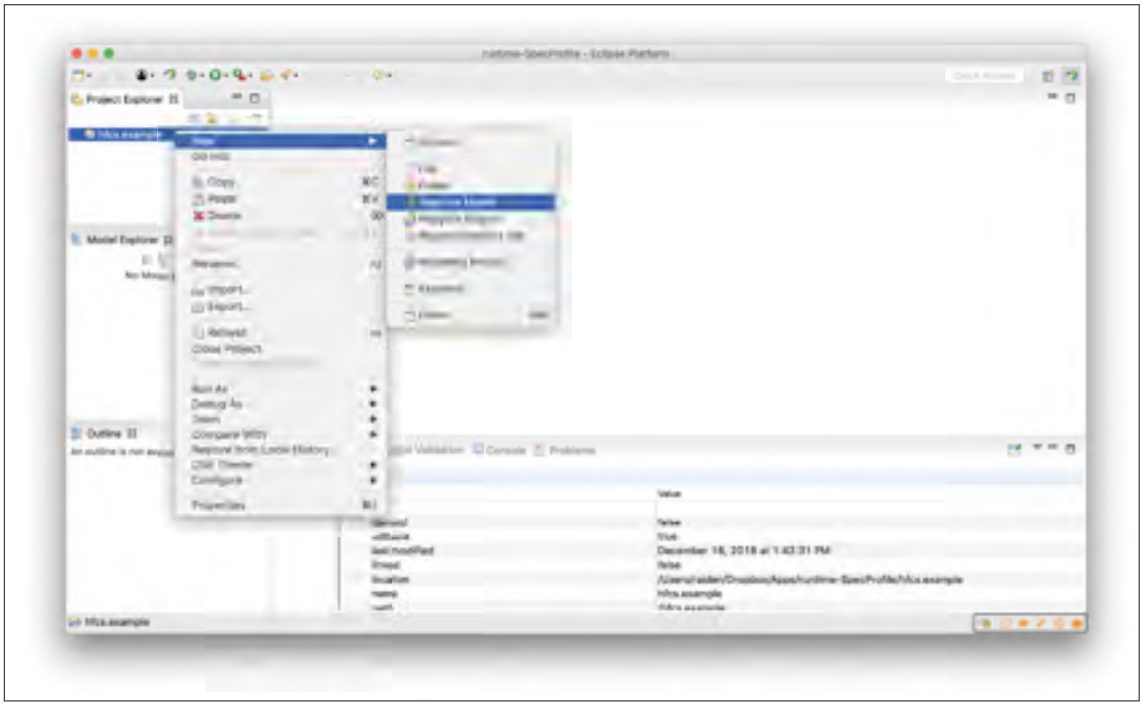
1. Open the Eclipse Papyrus modelling environment.



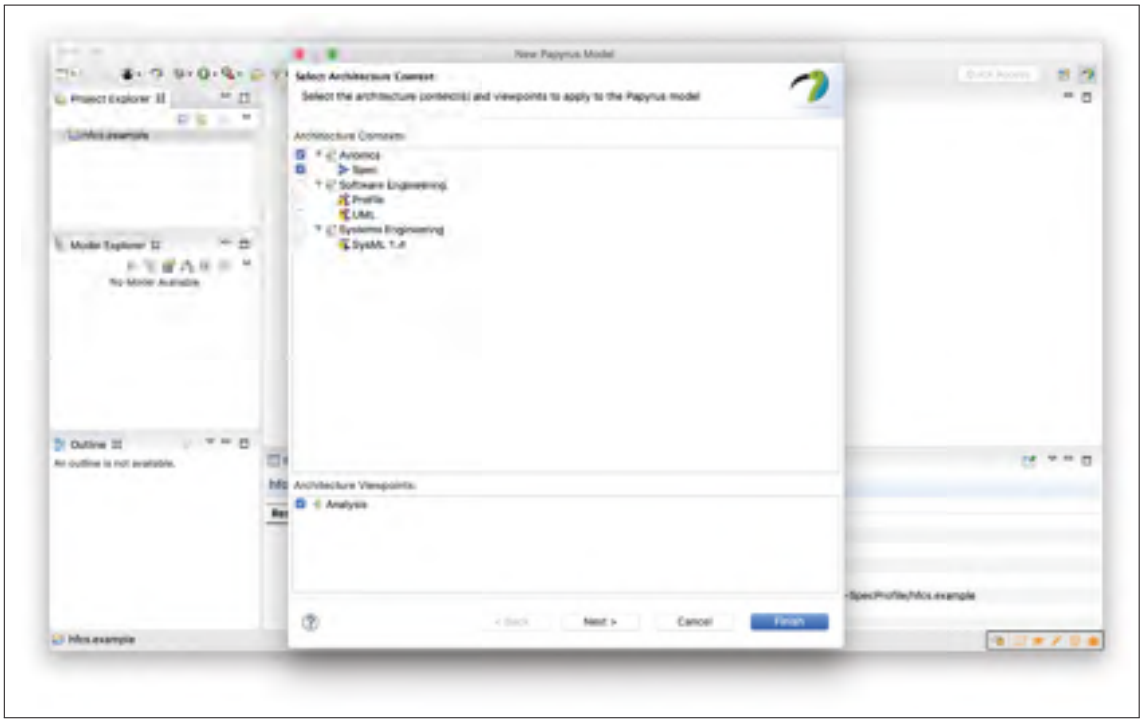
2. Create a project.



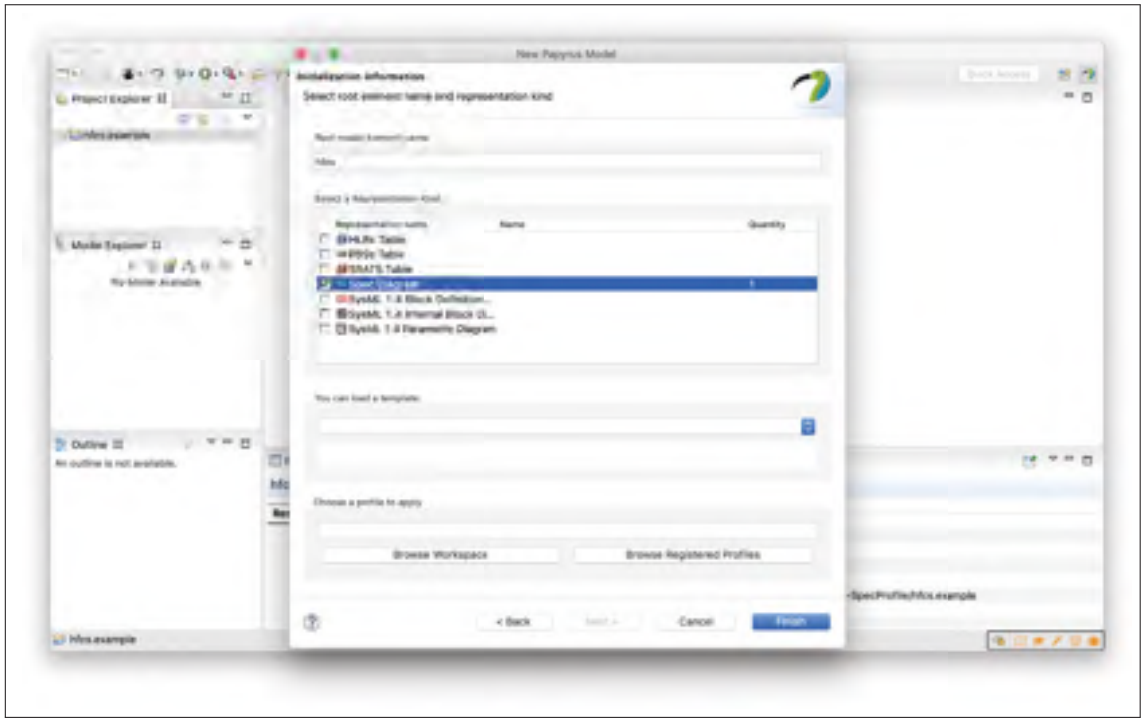
3. Create a new Papyrus model.



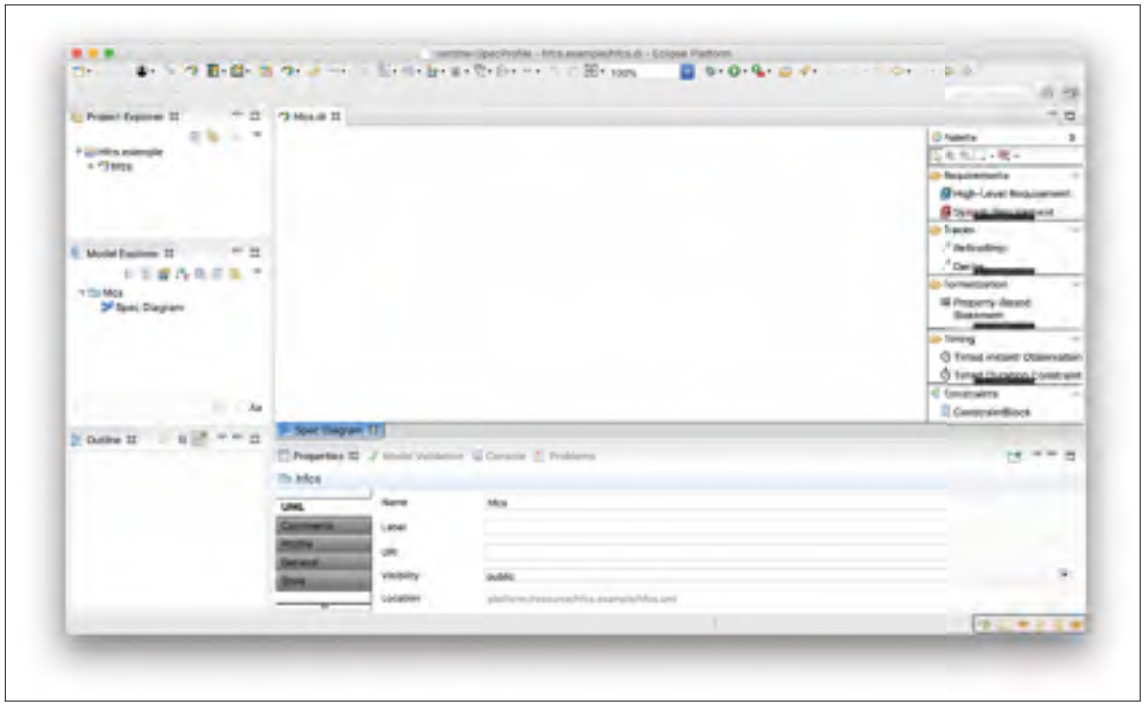
4. Select the SpecML architecture context.



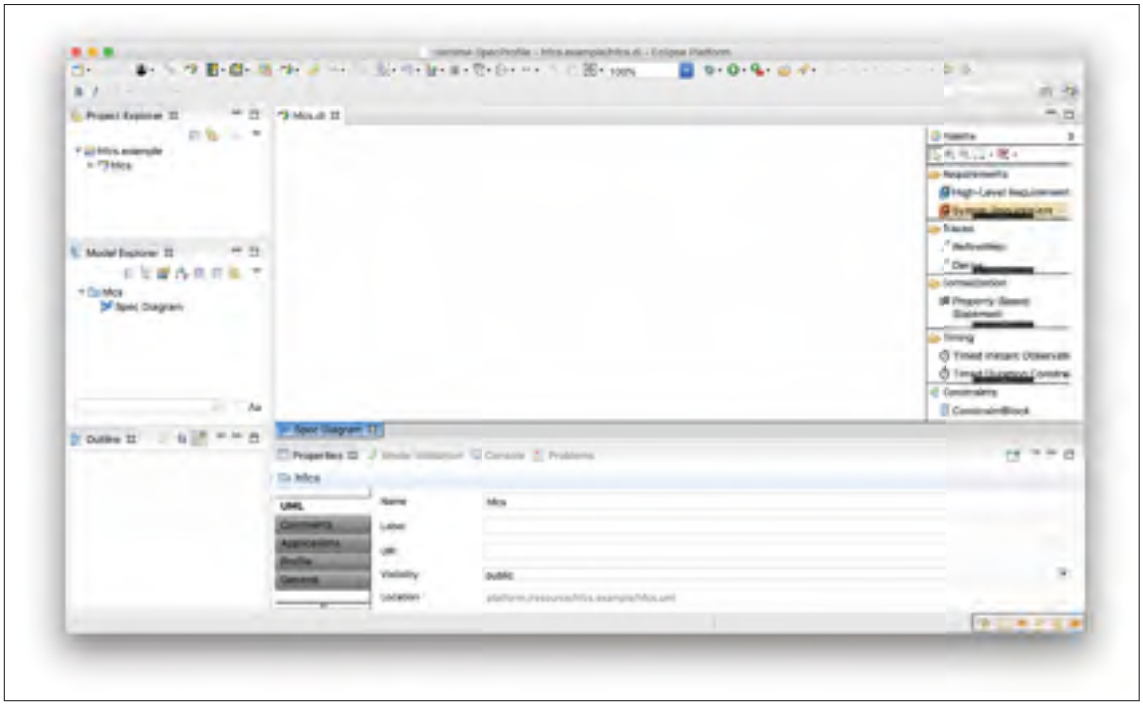
5. Select the SpecML diagram representation kind.



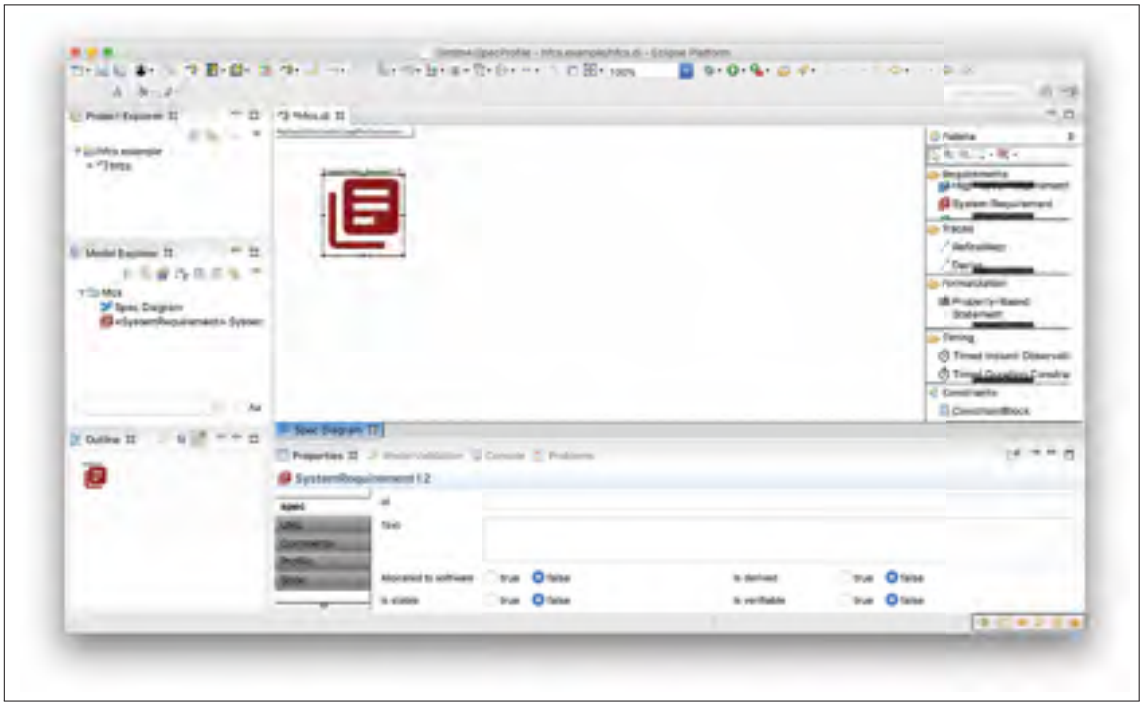
6. Click on **Finish**. The model should open.



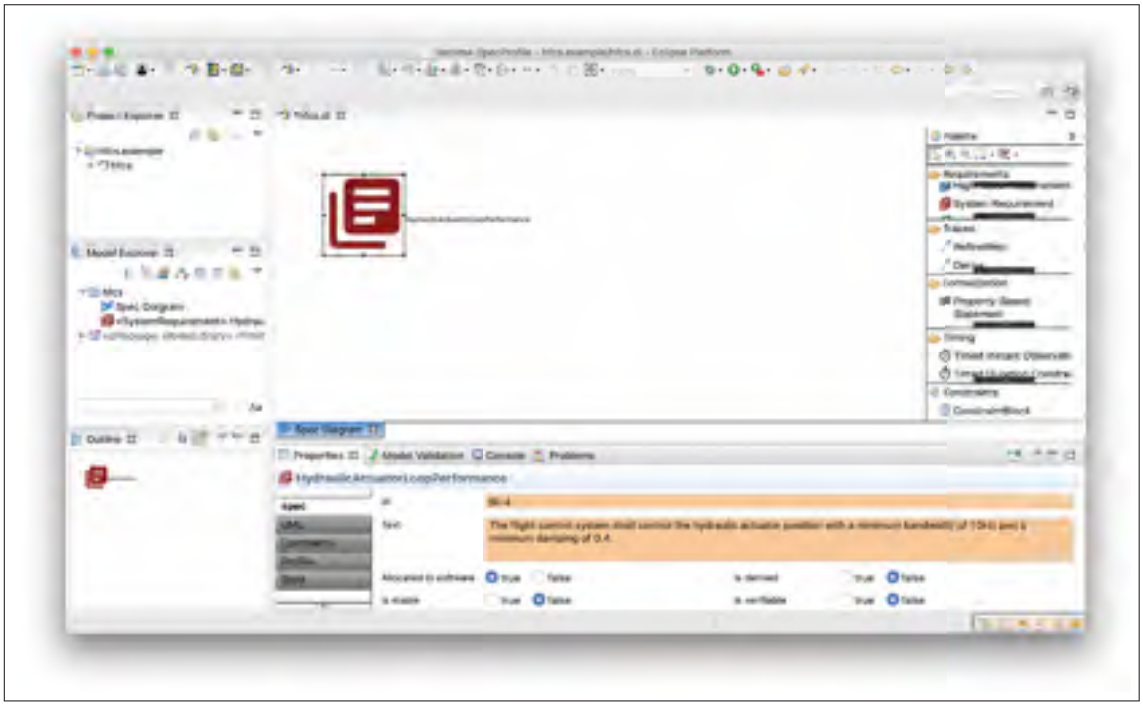
7. Drag and drop elements into the model by using the palette on the right. For instance, select the **System Requirement** element.



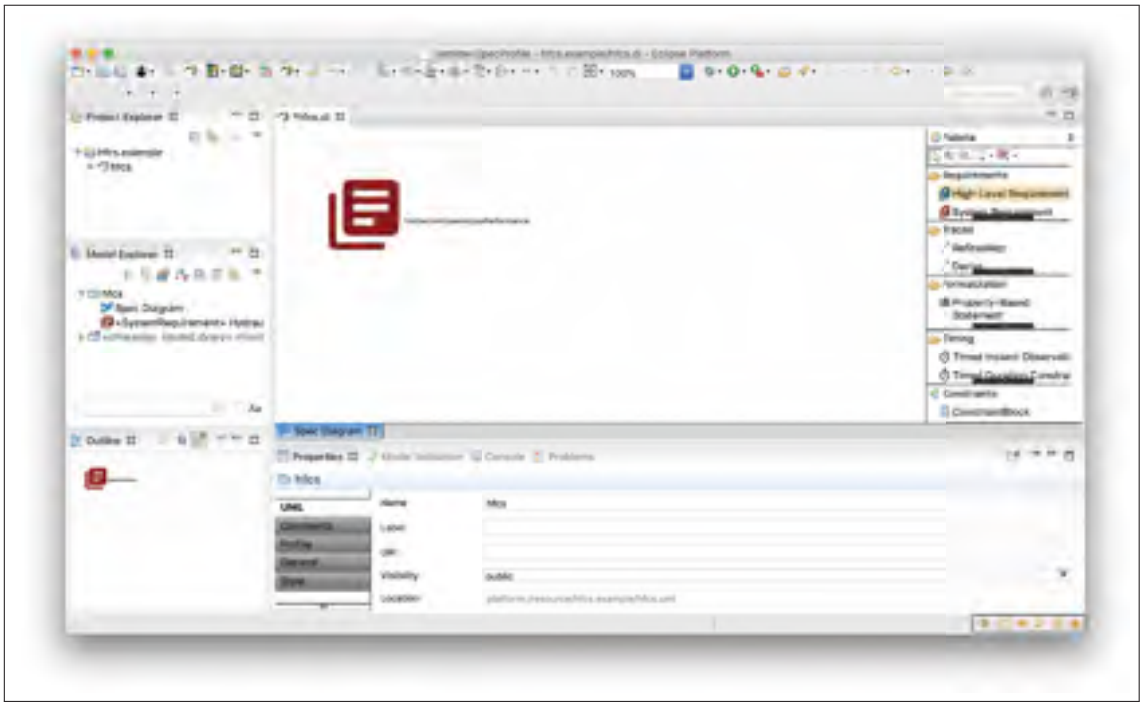
8. Drop the element anywhere in the model and give it a name.



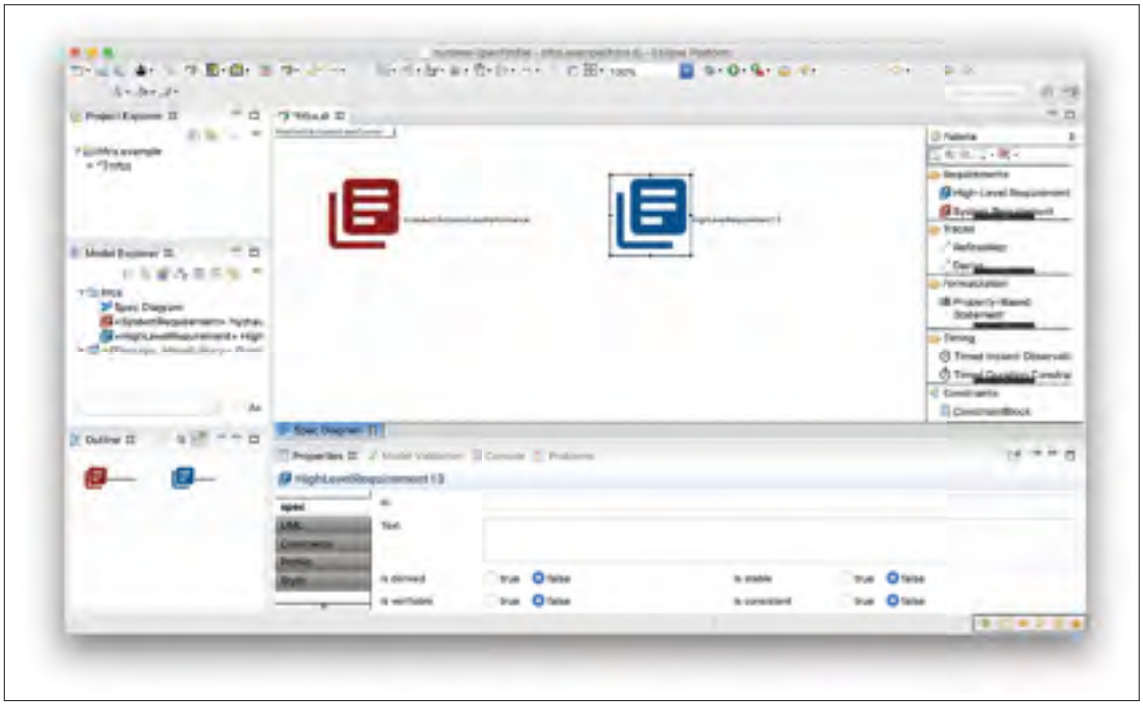
9. Fill out the properties of the element in the **Properties** view (bottom of the screen).



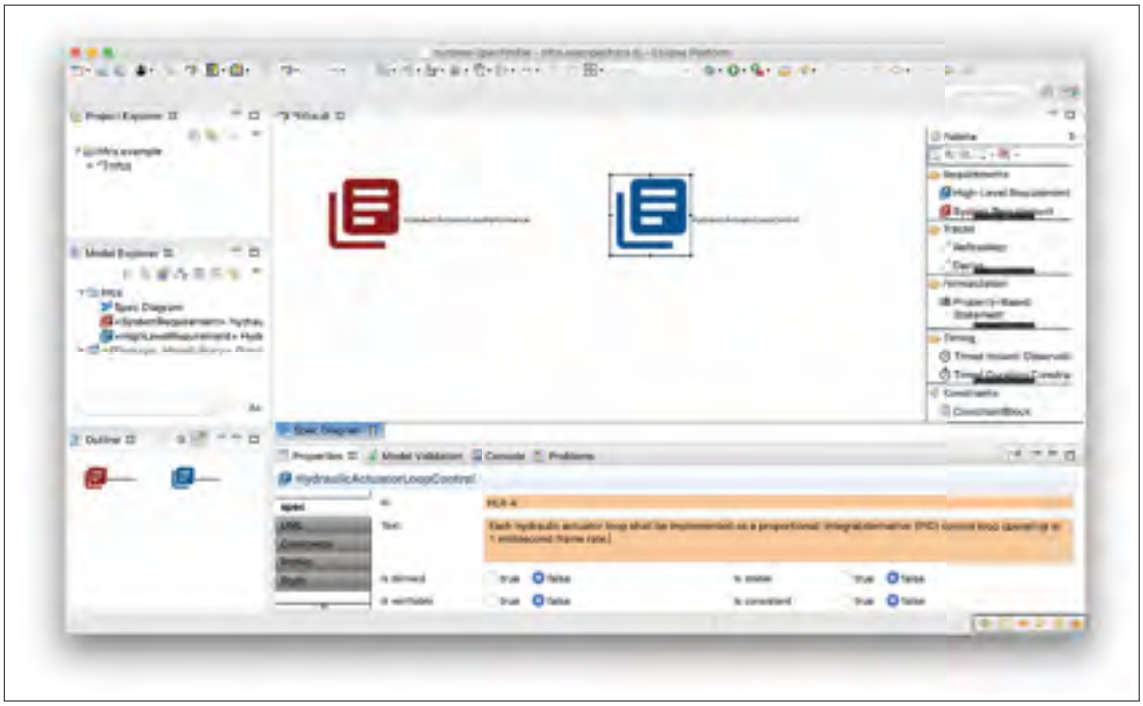
10. Add more elements into the model in the same way. For instance, select the **High-Level Requirement** element.



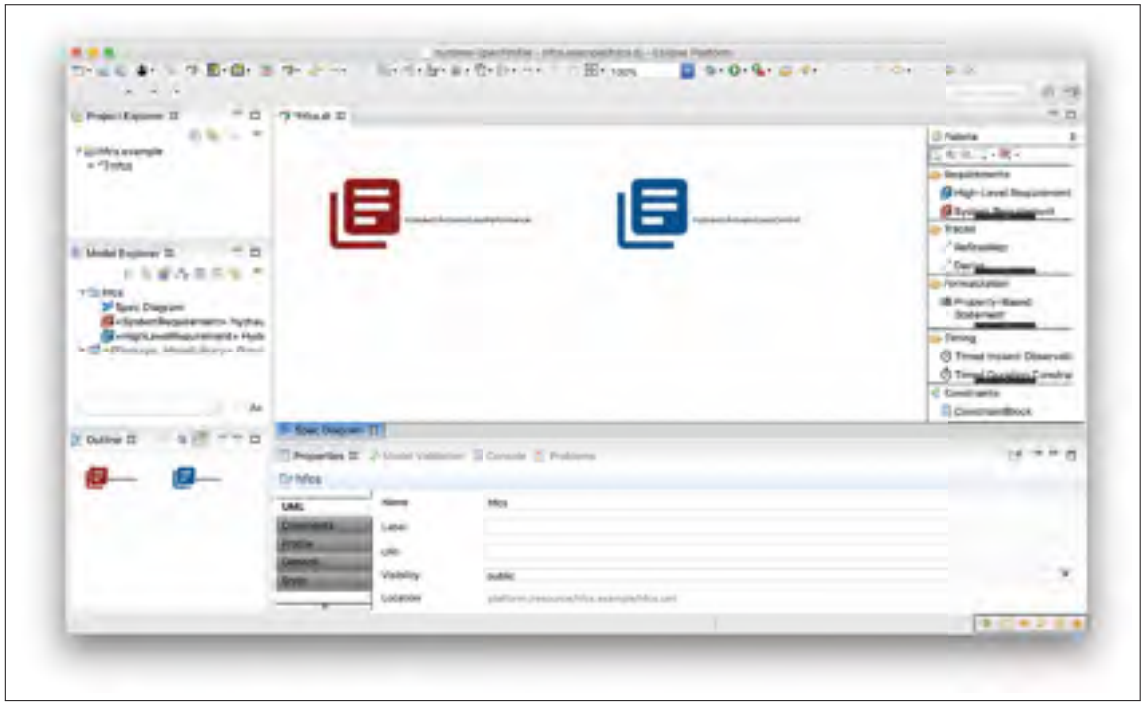
11. Drop the element anywhere in the model and give it a name.



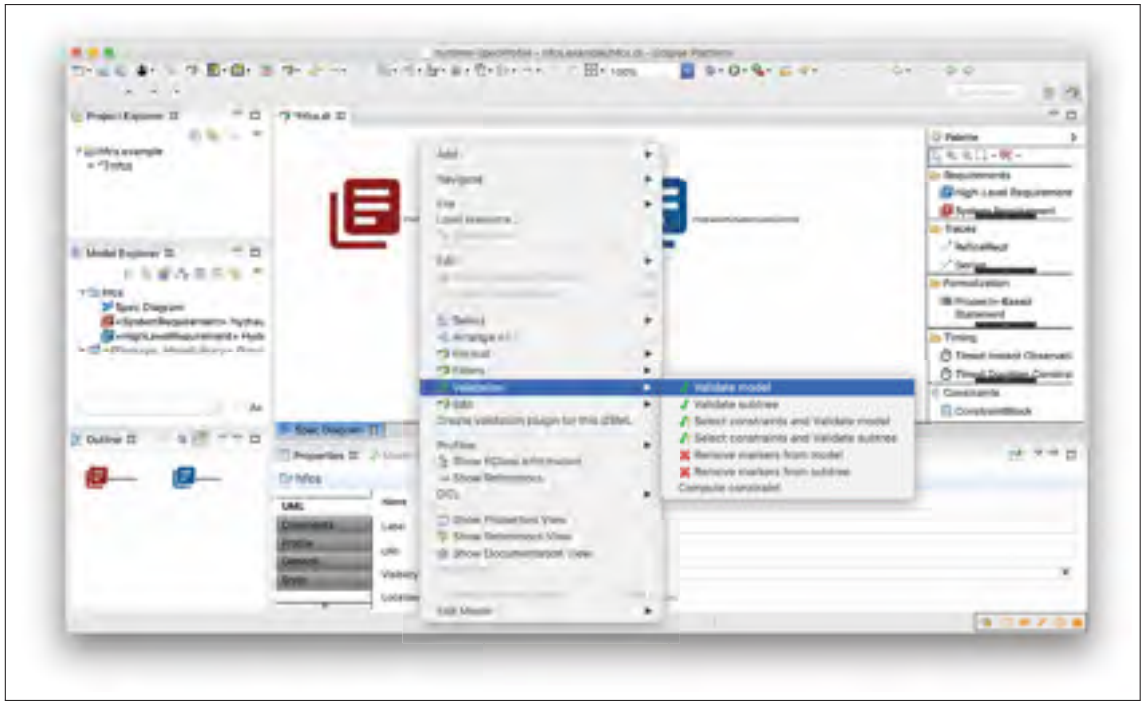
12. Fill out the properties of the element in the **Properties** view.



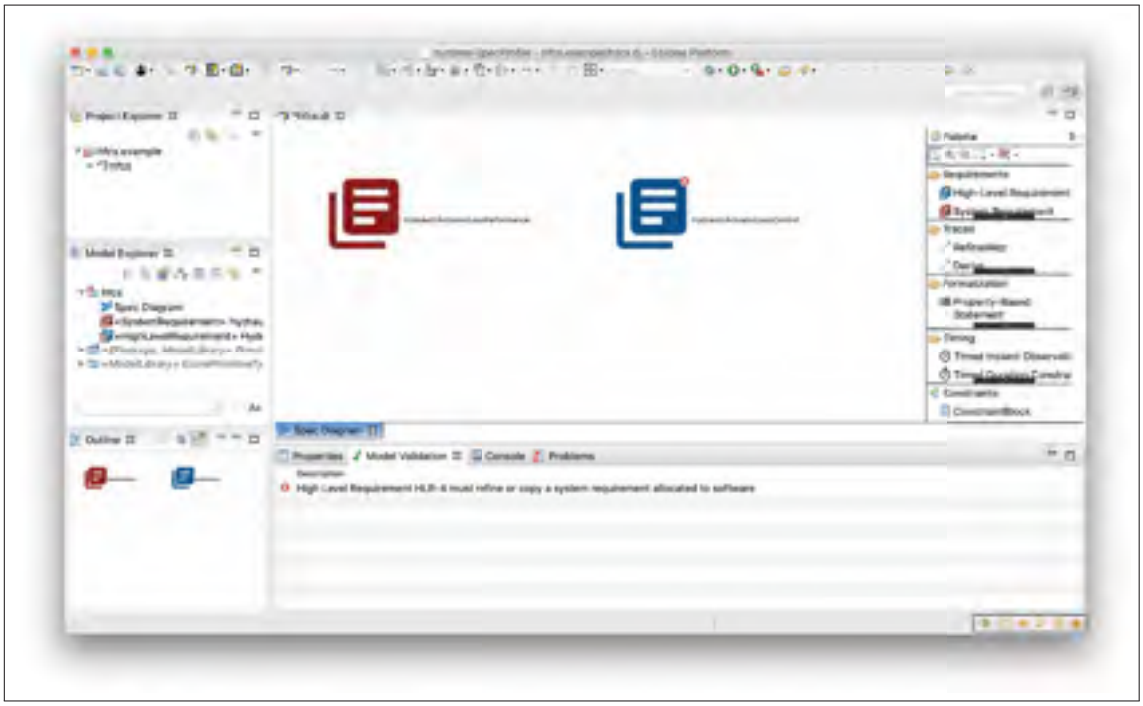
13. Continue adding more elements into the model.



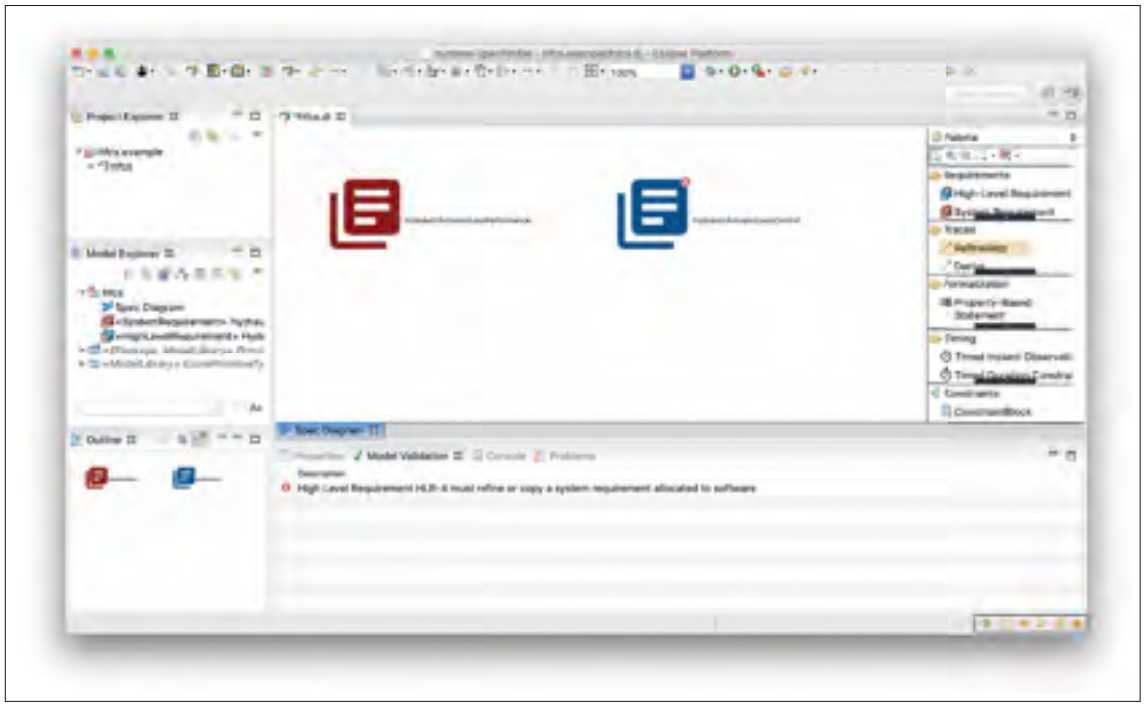
14. A verification of well-formedness of the model can be carried out at anytime. Right click anywhere in the model and select **Validation** » **Validate model**.



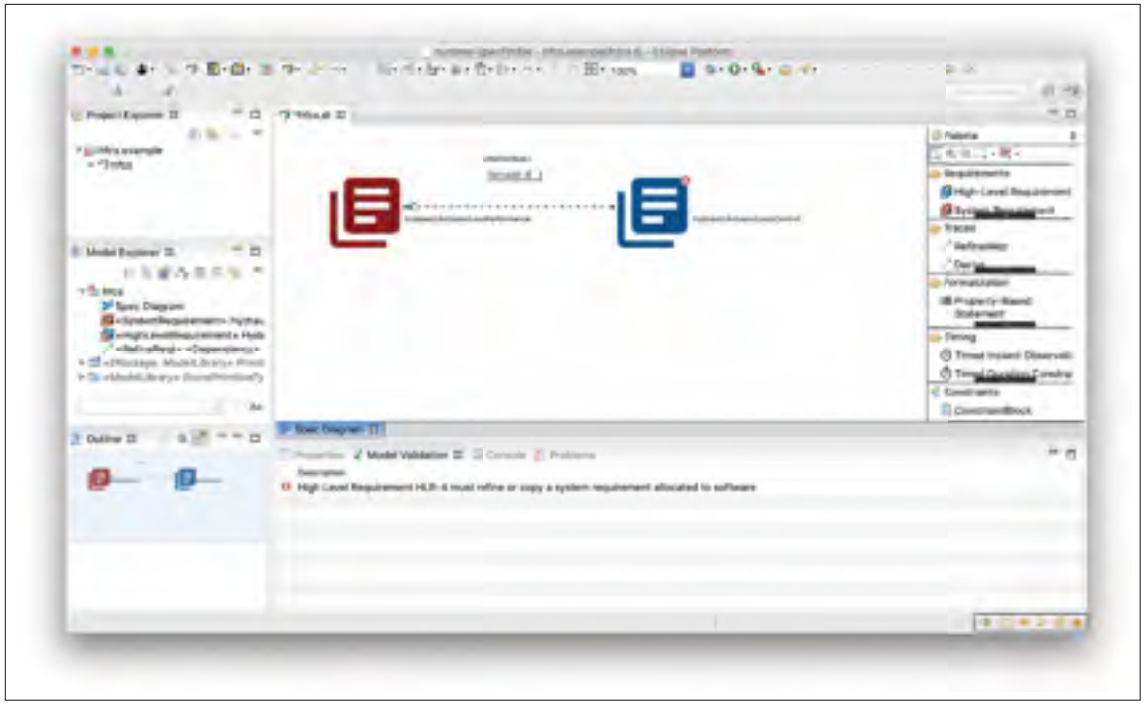
15. Violations will be presented in the **Model Validation** view (bottom of the screen). Any violating element will be marked in the editor view as well. In this case, the HLR (in blue) is not refining or copying an SRATS. To fix a violation, follow the indications given in the violation message.



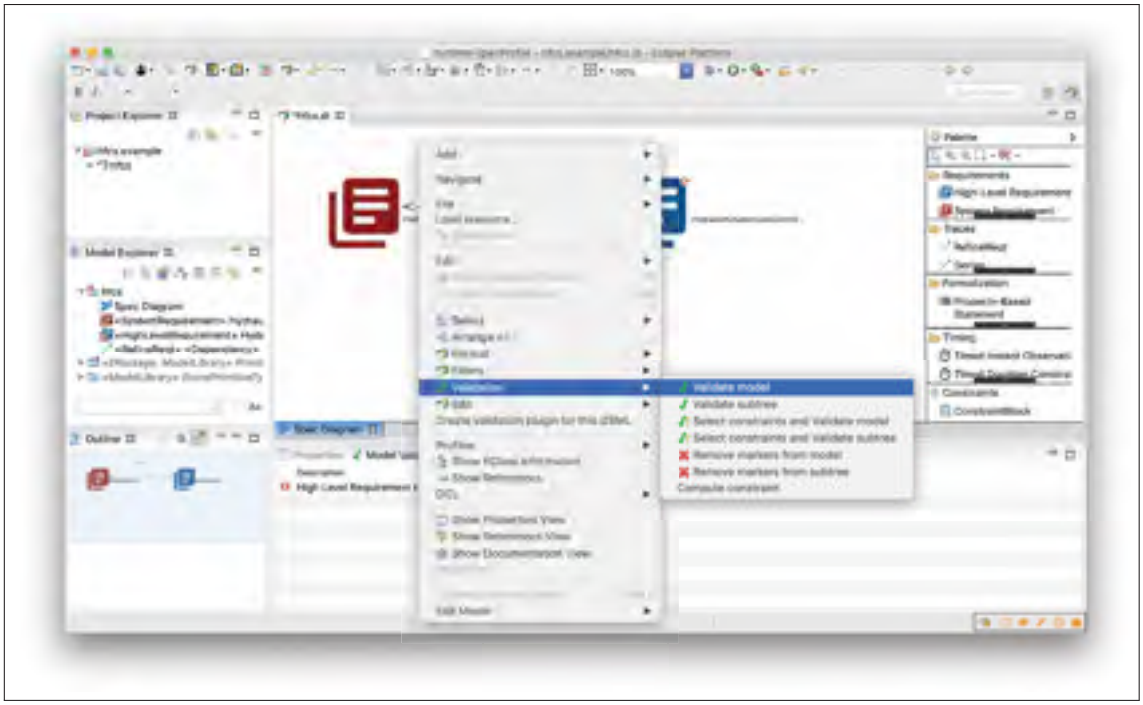
16. In this case, the violation will be fixed by defining a **RefineReqt** trace between the HLR (in blue) and the SRATS (in red).



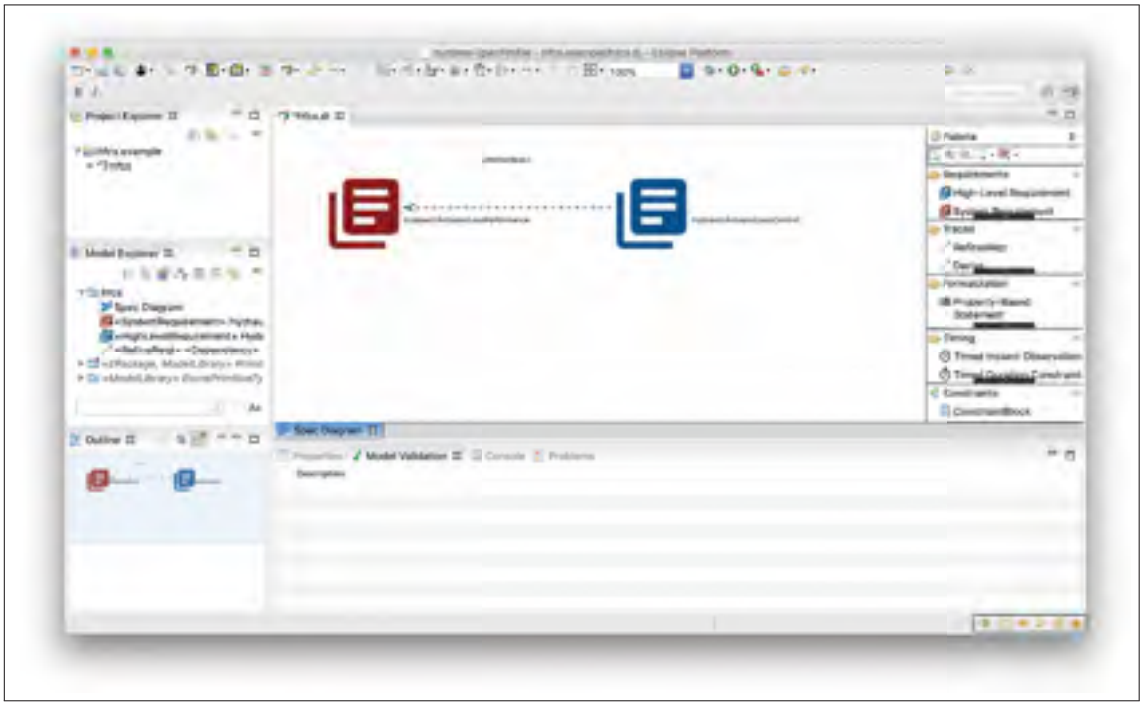
17. Click on the HLR to set the trace's client element. Then, click on the SRATS to set the trace's supplier element. Give the trace a name.



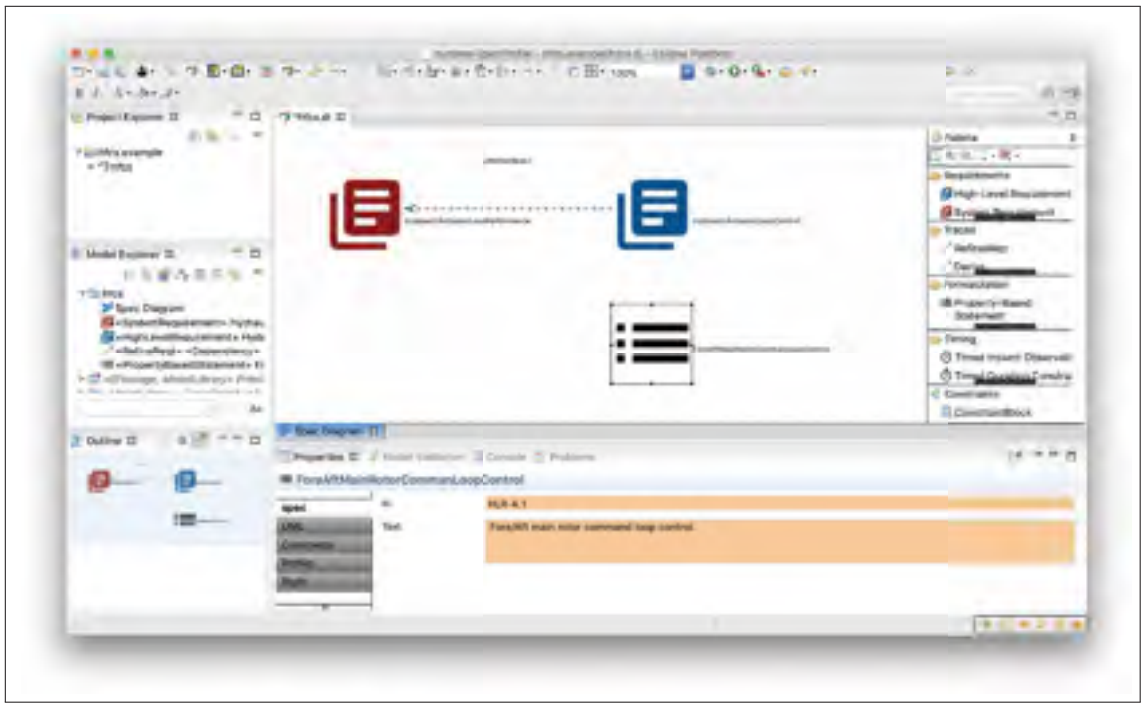
18. The violation was fixed. Re-verify the model by right clicking anywhere in the model and select **Validation** » **Validate model**.



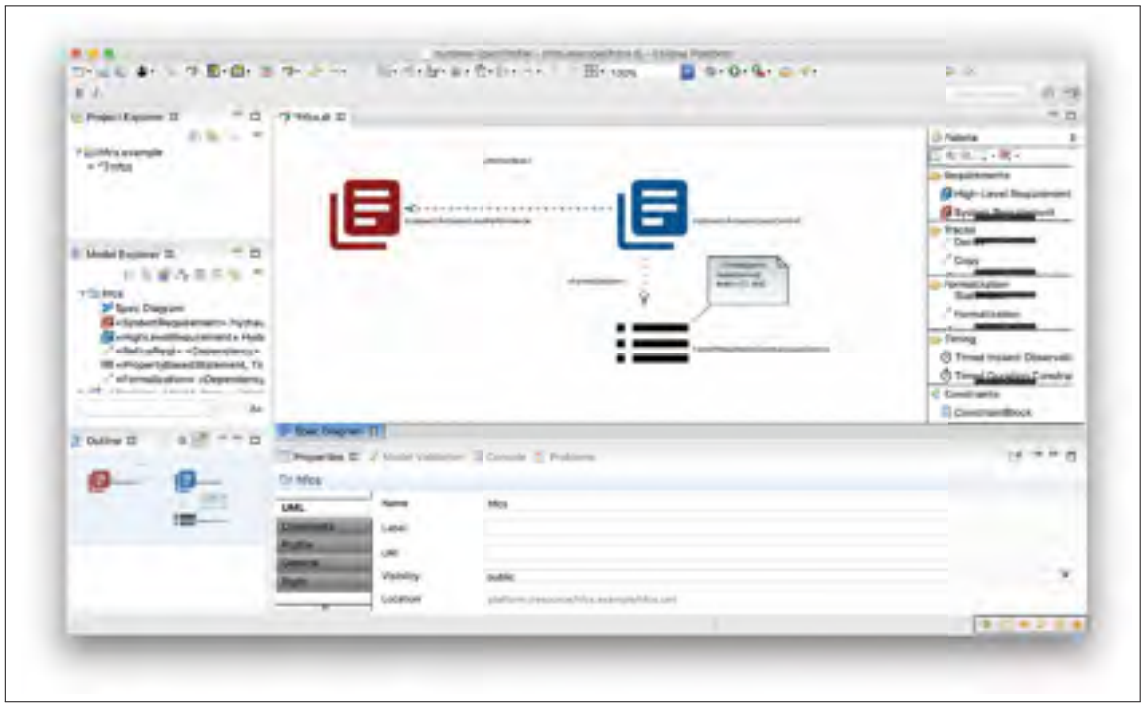
19. The violation should disappear from the **Model Validation** view.



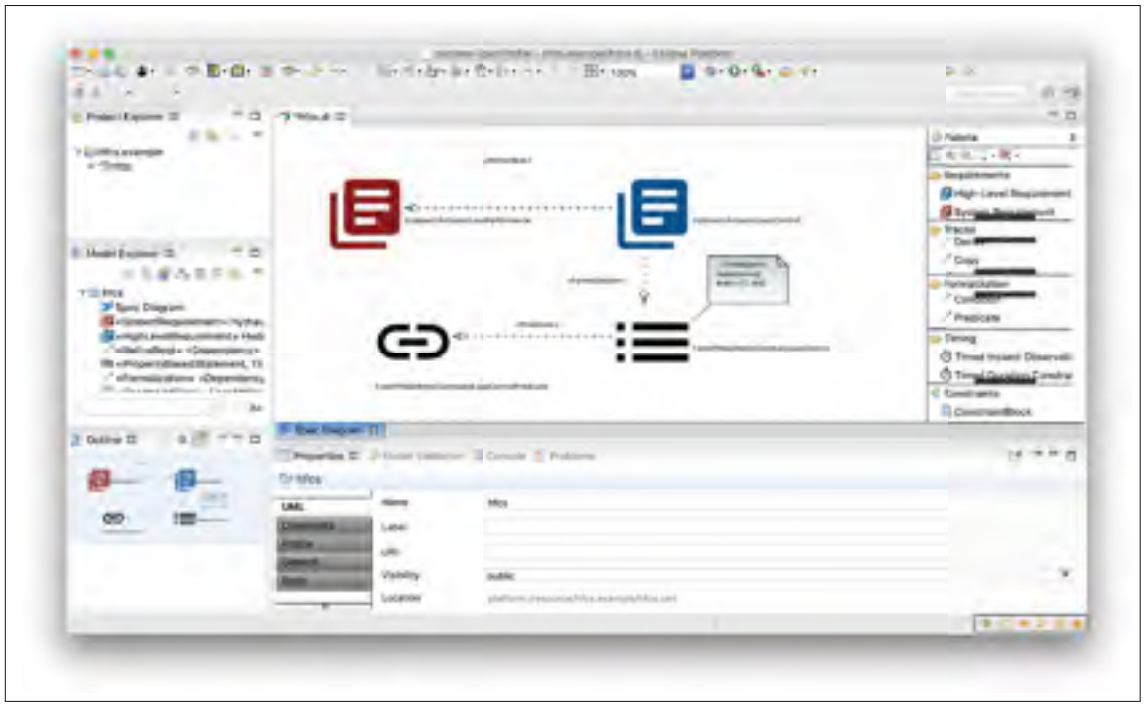
20. Continue adding more elements into the model. For instance, drag and drop a **Property-Based Statement** element and give it a name. Fill out its properties in the **Properties** view.



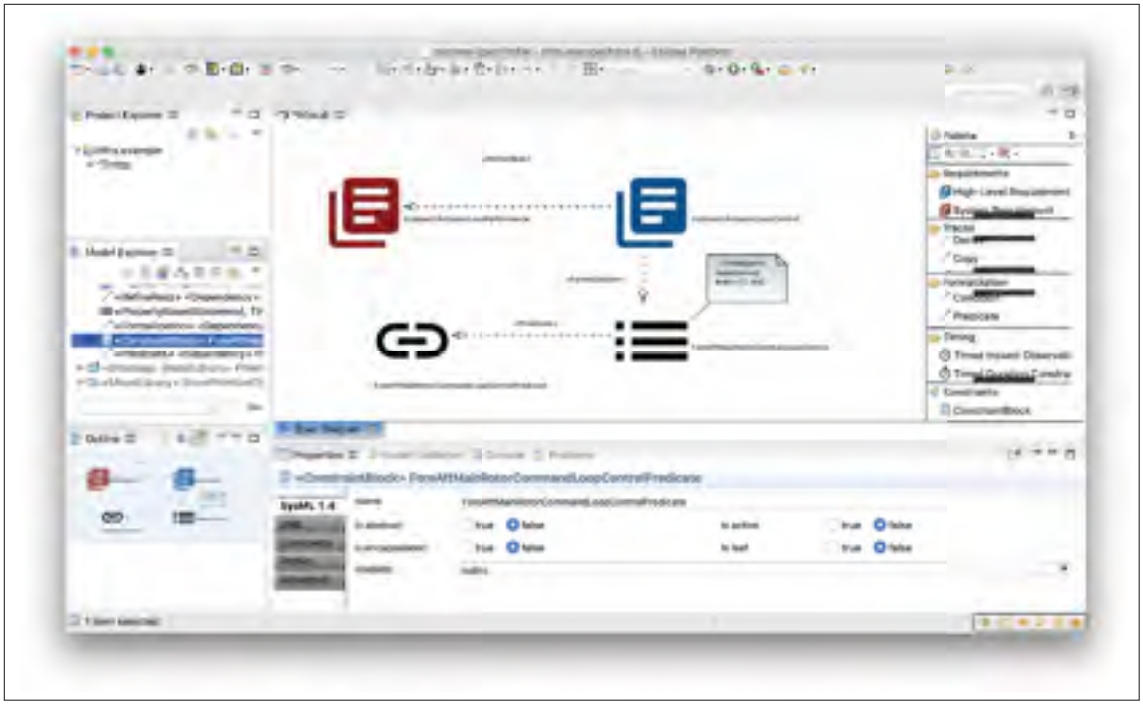
21. Define time-dependent behaviour and constraints by applying MARTE stereotypes onto existing property-based statements or dragging and dropping standalone elements from the palette. In this case, the **TimedEvent** stereotype is applied on the **Property-Based Statement** and displayed as a comment. This is done in the **Profile** and **Appearance** tabs of the **Properties** view (respectively).



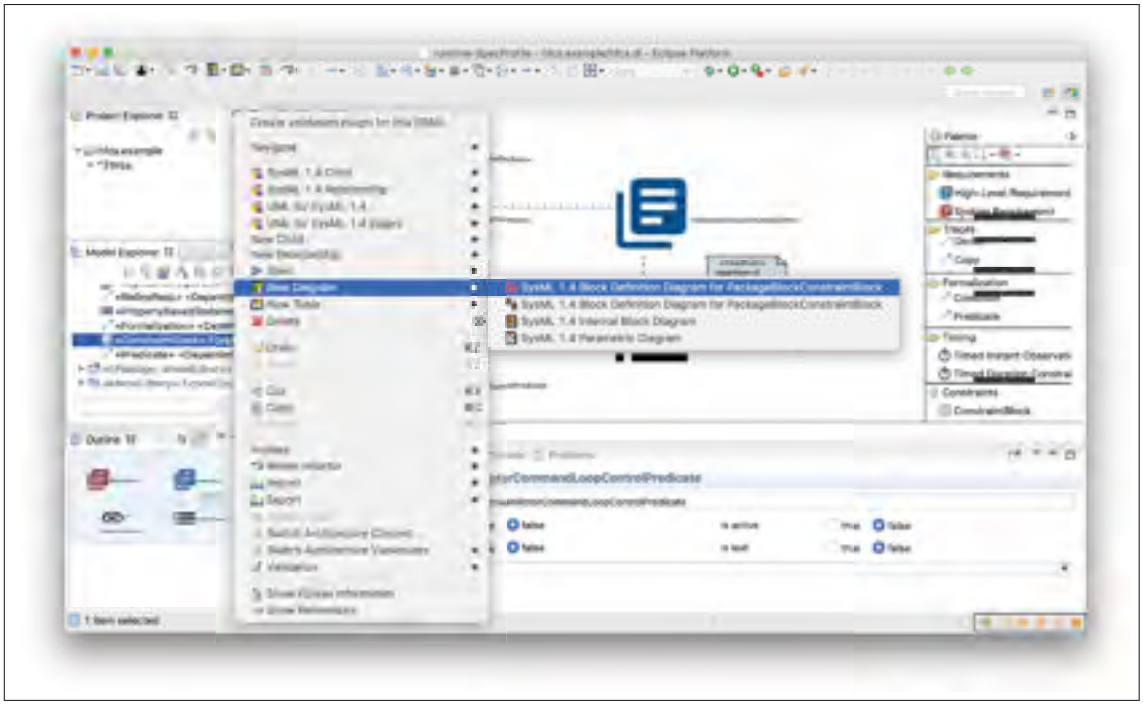
22. Add a **ConstraintBlock** and define as the predicate of the **Property-Based Statement** with the **Predicate** formalization trace.



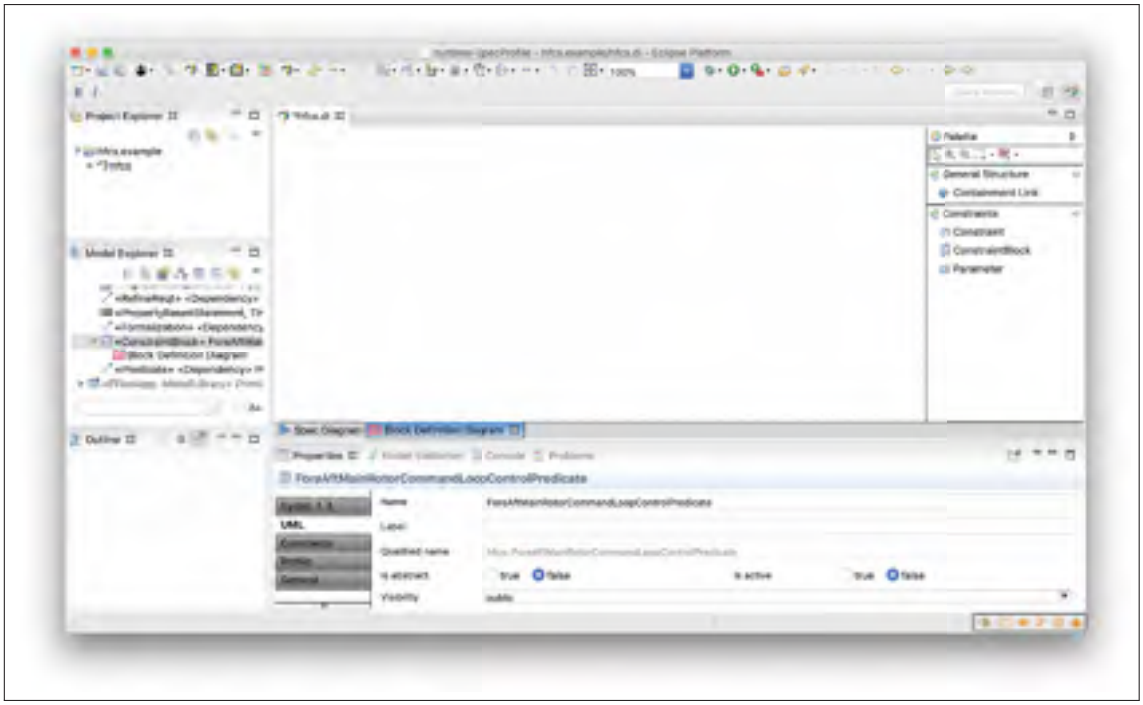
23. In this case, the predicate of this statement has nested SysML constraint blocks. This constraint blocks need to be defined in a SysML block diagram. Select the predicate constraint block in the **Model Explorer** view (left center of the screen).



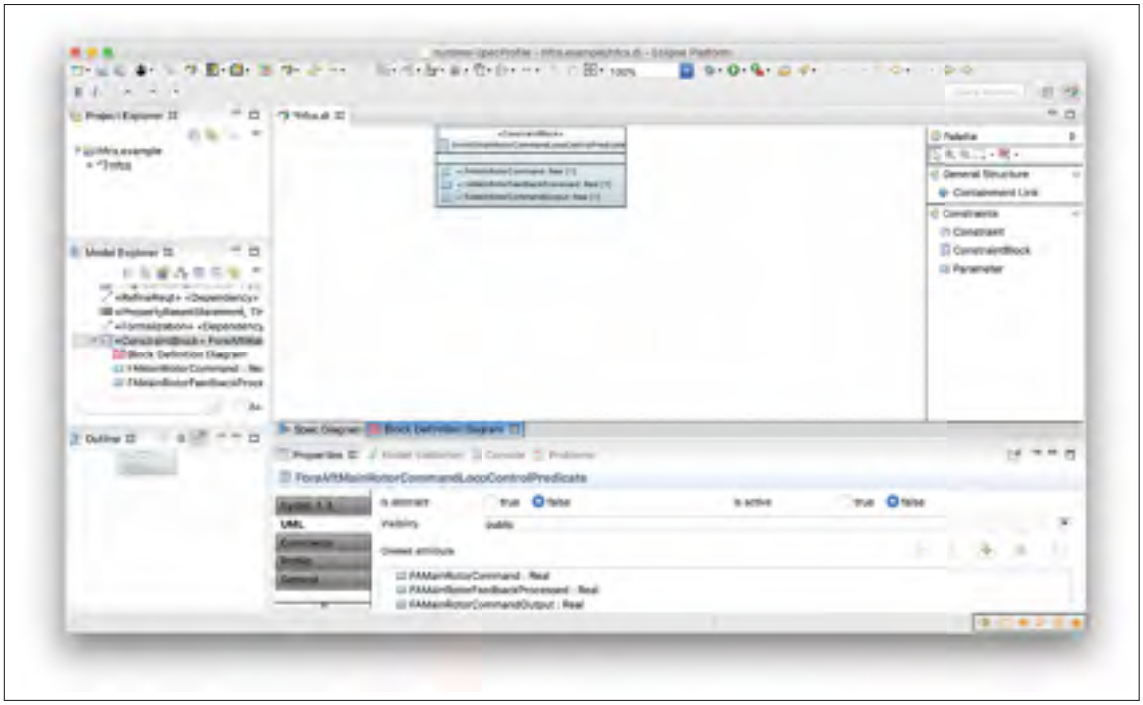
24. Right click and select **New Diagram** » **SysML 1.4 Block Definition Diagram** (first option).



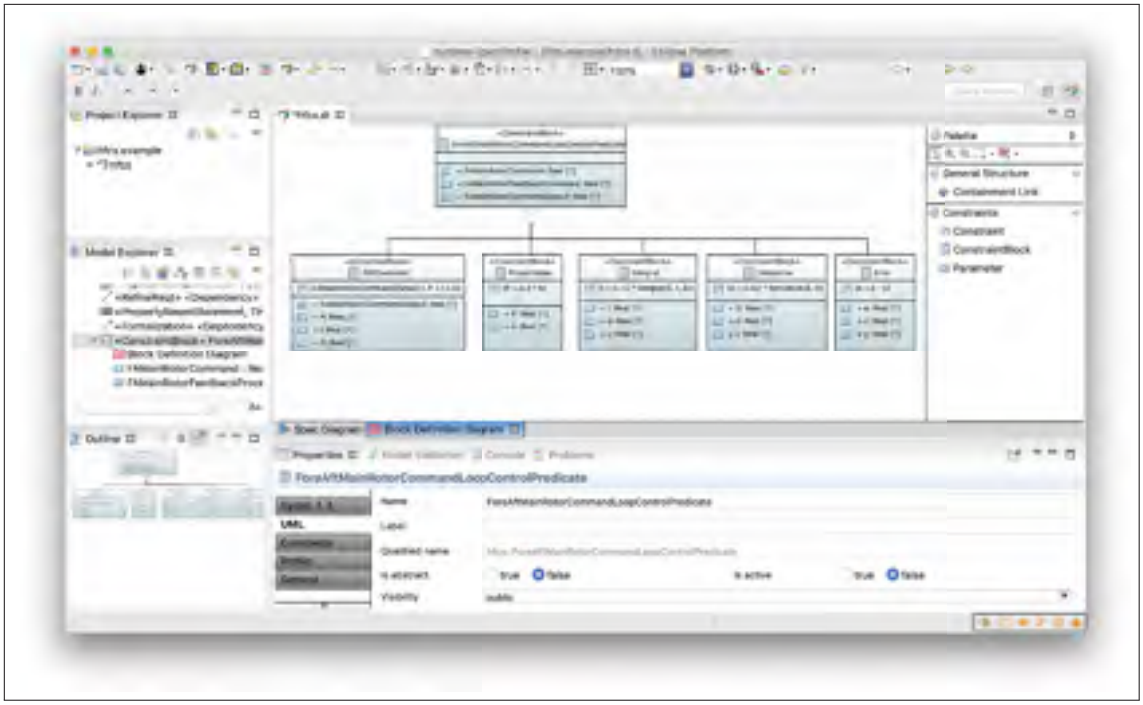
25. A new tab in the model is created.



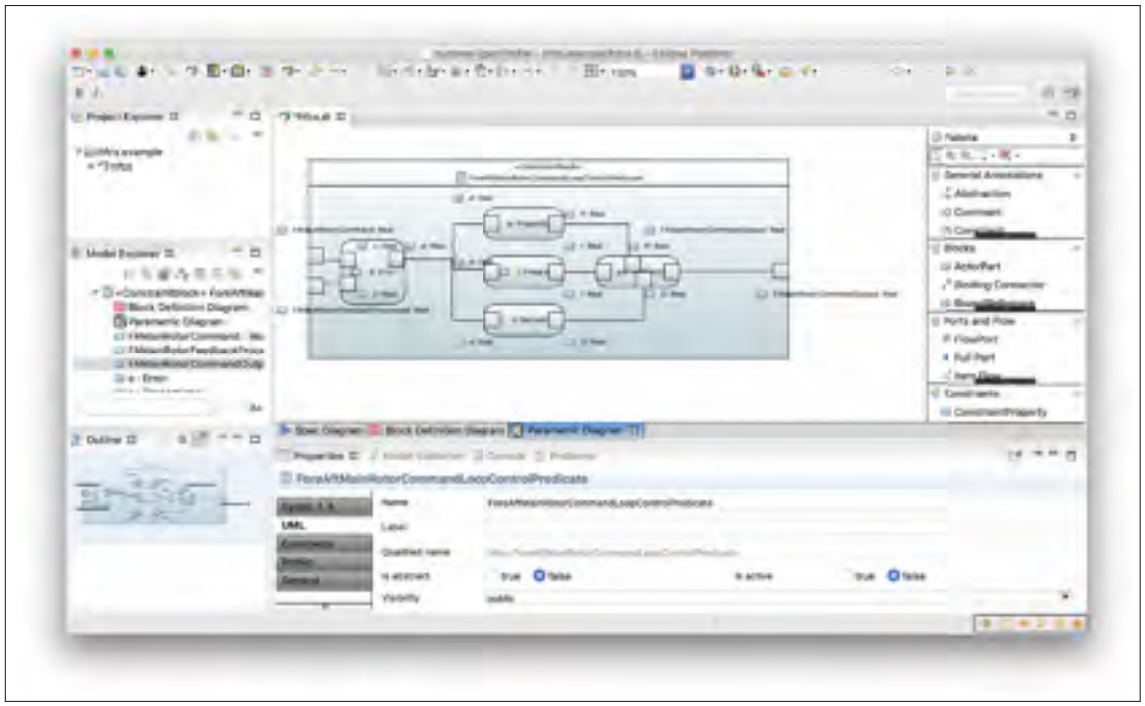
26. Drag and drop the selected constraint block of the predicate from the **Model Explorer** view.



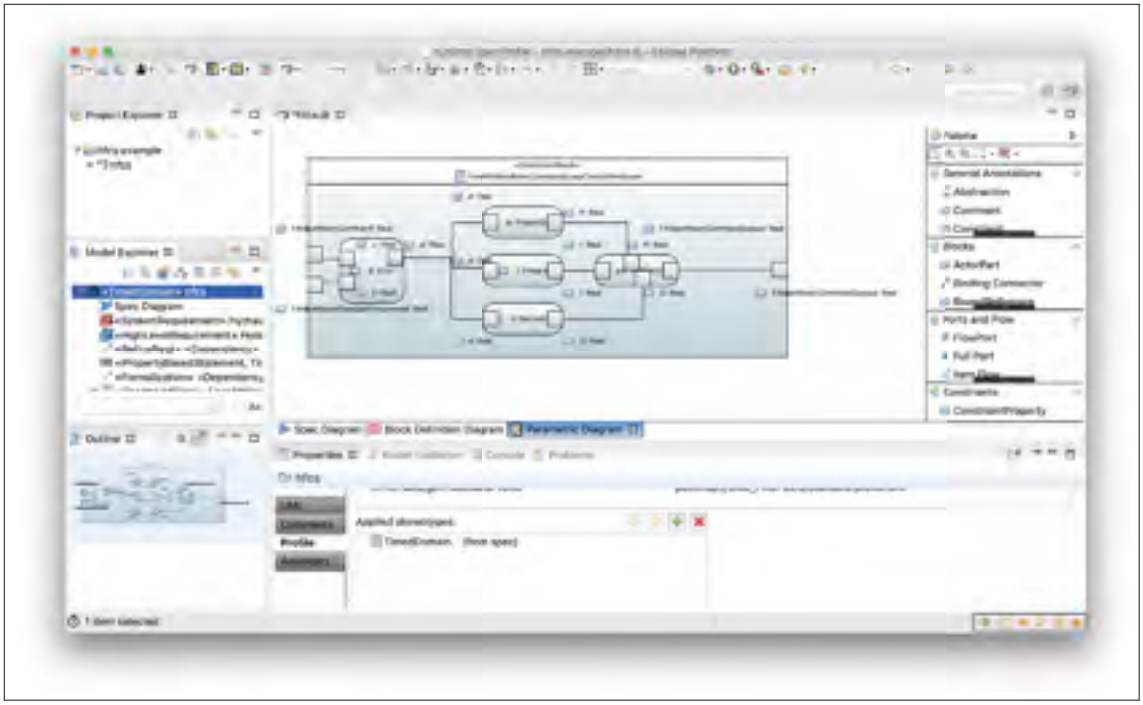
27. Add the nested constraint blocks by dragging and dropping from the palette and connecting them with the **Containment Link**.



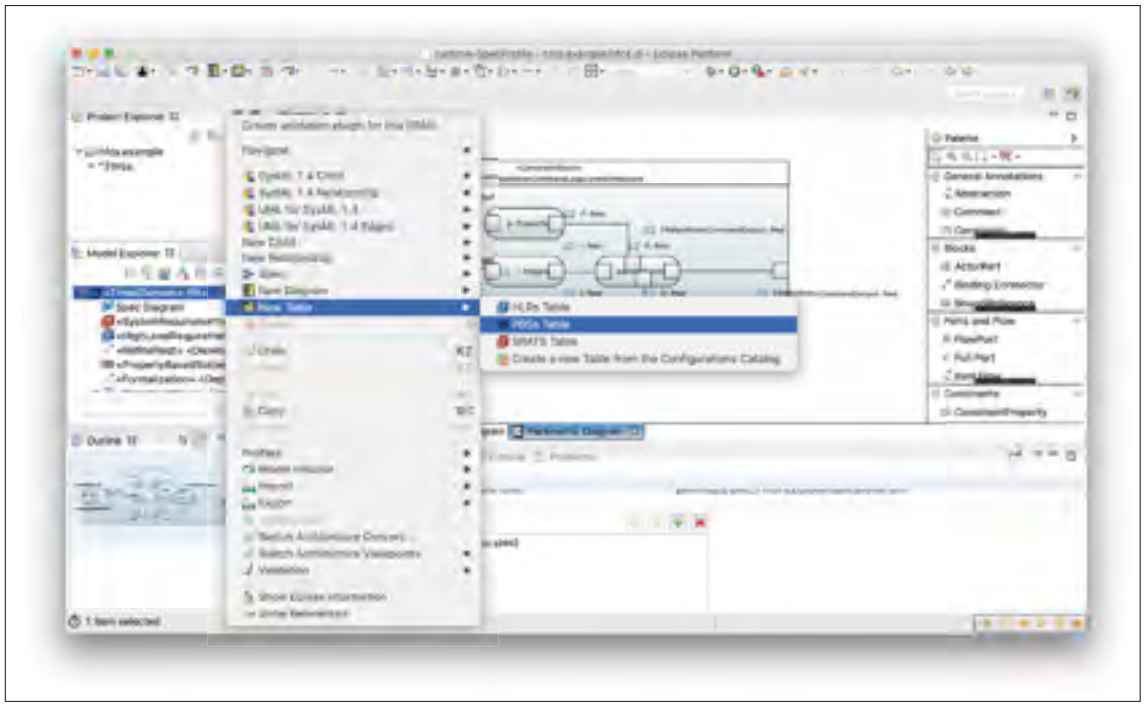
28. In a similar way, a parametric diagram can be added to detail the usage of the nested constraints.



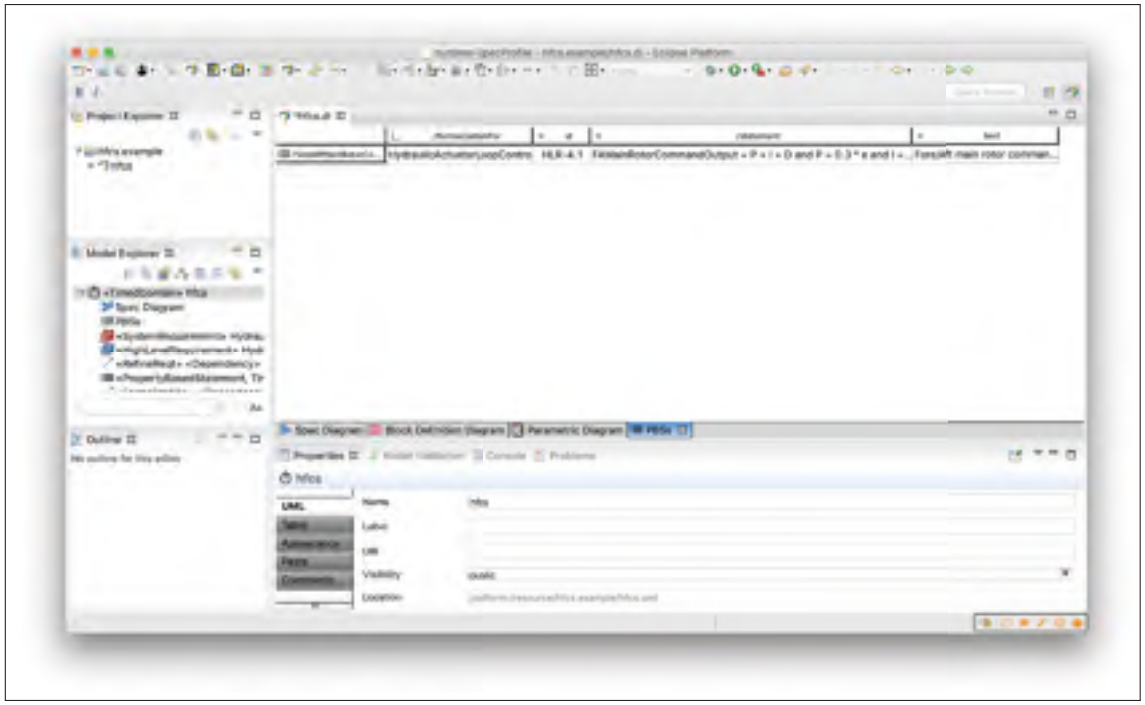
29. If there is time-dependent behaviour and constraints, the specification model must be stereotyped with **TimedDomain**. This is done in the **Profile** tab of the **Properties** view with the SpecML diagram selected in the **Model Explorer** view.



30. The specified requirements and their formalizations can be viewed in tables. Right click and select, for instance, **New Table** » **PBSs Table**.



31. In this case, a table with the requirement formalizations is created.



BIBLIOGRAPHY

acatech (Ed.). (2011). *Cyber-Physical Systems: Driving Force for Innovation in Mobility, Health, Energy and Production*. Berlin, Heidelberg: Springer-Verlag.

Almeida da Silva, M. A., Mougenot, A., Blanc, X. & Bendraou, R. (2010). Towards Automated Inconsistency Handling in Design Models. *Advanced Information Systems Engineering*, pp. 348–362.

Amyot, D. (2003). Introduction to the User Requirements Notation: Learning by Example. *Computer Networks*, 42(3), 285–301.

Arboleda, H. & Royer, J.-C. (2012). *Model-Driven and Software Product Line Engineering*. London: ISTE and Wiley.

Areias, C., Cunha, J. C., Iacono, D. & Rossi, F. (2014). Towards Certification of Automotive Software. *2014 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 491-496.

Basili, V. R., Caldiera, G. & Rombach, H. D. (1994). The Goal Question Metric Approach. In *Encyclopedia of Software Engineering*. Wiley.

Berkenkötter, K. & Hannemann, U. (2006). Modeling the Railway Control Domain Rigorously with a UML 2.0 Profile. In *Proceedings of the 25th International Conference on Computer Safety, Reliability, and Security (SAFECOMP)* (pp. 398–411). Berlin, Heidelberg: Springer Berlin Heidelberg.

Bézivin, J. (2005). On the unification power of models. *oftware & Systems Modeling*, 4(2), 171–188.

Bézivin, J. (2006). Model Driven Engineering: An Emerging Technical Space. In *Generative and Transformational Techniques in Software Engineering* (pp. 36–64). Berlin - Heidelberg, Germany: Springer.

Bialy, M., Lawford, M., Pantelic, V. & Wassyng, A. (2015, 5). A Methodology for the Simplification of Tabular Designs in Model-Based Development. *2015 IEEE/ACM 3rd FME Workshop on Formal Methods in Software Engineering (FormaliSE)*, pp. 47–53.

Bialy, M., Pantelic, V., Jaskolka, J., Schaap, A., Patcas, L., Lawford, M. & Wassyng, A. (2017). 3 - Software Engineering for Model-Based Development by Domain Experts. In Griffor, E. (Ed.), *Handbook of System Safety and Security* (pp. 39–64). Boston: Syngress.

Biggs, G., Sakamoto, T. & Kotoku, T. (2016). A profile and tool for modelling safety information with design information in SysML. *Software & Systems Modeling*, 15(1), 147–178.

Bitsch, F. (2001). Safety Patterns — The Key to Formal Specification of Safety Requirements. In *Computer Safety, Reliability and Security: 20th International Conference, SAFECOMP 2001 Budapest, Hungary, September 26–28, 2001 Proceedings* (pp. 176–189). Berlin, Heidelberg: Springer Berlin Heidelberg.

Blouin, D., Senn, E. & Turki, S. (2011). Defining an annex language to the architecture analysis and design language for requirements engineering activities support. *2011 Model-Driven Requirements Engineering Workshop*, pp. 11–20.

Blouin, D. (2013). *Modeling languages for requirements engineering and quantitative analysis of embedded systems*. (Ph.D. thesis, Université de Bretagne-Sud).

Bombino, M. & Scandurra, P. (2013). A model-driven co-simulation environment for heterogeneous systems. *International Journal on Software Tools for Technology Transfer*, 15(4), 363–374.

Boniol, F. & Wiels, V. (2014). The Landing Gear System Case Study. In *ABZ 2014: The Landing Gear Case Study*. Springer.

Bozga, M., Graf, S., Ober, I., Ober, I. & Sifakis, J. (2004). The IF Toolset. In Bernardo, M. & Corradini, F. (Eds.), *Formal Methods for the Design of Real-Time Systems: International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Revised Lectures* (pp. 237–267). Berlin, Heidelberg: Springer Berlin Heidelberg.

Bozzano, M. & Villafiorita, A. (2010). *Design and Safety Assessment of Critical Systems* (ed. 1). Auerbach Publications.

Camhy, D., Kernbichler, W., Huhs, G. & Albert, C. (2013). Matclipse – Eclipse-Matlab interface. Consulted at https://undocumentedmatlab.com/blog/matclipse-eclipse-matlab-interface.

Carnegie Mellon University. (2019). OSATE (Open Source AADL Tool Environment). Consulted at https://osate.org/.

Ceccarelli, A. & Silva, N. (2013). Qualitative comparison of aerospace standards: An objective approach. *2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 331–336.

Combemale, B., DeAntoni, J., Baudry, B., France, R. B., Jézéquel, J. & Gray, J. (2014). Globalizing Modeling Languages. *Computer*, 47(6), 68–71.

Creswell, J. W. (2008). *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches* (ed. 3). Sage Publications Ltd.

de la Vara, J. L., Ruiz, A., Attwood, K., Espinoza, H., P.-W., R. K., Ángel López, del Río, 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.

Dijkman, R. M., Quartel, D. A. & van Sinderen, M. J. (2008). Consistency in multi-viewpoint design of enterprise information systems. *Information and Software Technology*, 50(7), 737–752.

Easterbrook, S., Singer, J., Storey, M.-A. & Damian, D. (2008). In Shull, F., Singer, J. & Sjøberg, D. I. K. (Eds.), *Guide to Advanced Empirical Software Engineering* (ch. Selecting Empirical Methods for Software Engineering Research, pp. 285–311). Springer London.

Eclipse Lyo. (2014). Lyo OSLC Simulink Adapter. Consulted at https://wiki.eclipse.org/Lyo/Simulink.

Eisemann, U. (2016, 01). Applying Model-Based Techniques for Aerospace Projects in Accordance with DO-178C, DO-331, and DO-333. *8th European Congress on Embedded Real Time Software and Systems (ERTS)*.

Eker, J., Janneck, J., Lee, E., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S. & Xiong, Y. (2003). Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1), 127–144.

El Hamlaoui, M., Bennani, S., Nassar, M., Ebersold, S. & Coulette, B. (2018). Heterogeneous Design Models Alignment: From Matching to Consistency Management. *Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC)*, pp. 1695–1697.

Erkkinen, T. (2005). Model Style Guidelines for Flight Code Generation. In *AIAA Modeling and Simulation Technologies Conference and Exhibit* (pp. 1–8). American Institute of Aeronautics and Astronautics, Inc.

Esposito, C., Cotroneo, D. & SIlva, N. (2011). Investigation on Safety-Related Standards for Critical Systems. *2011 First International Workshop on Software Certification*, pp. 49–54.

Farkas, T., Neumann, C. & Hinnerichs, A. (2009). An Integrative Approach for Embedded Software Design with UML and Simulink. *33rd Annual IEEE International Computer Software and Applications Conference*, 2, 516–521.

Fecher, H., Schönborn, J., Kyas, M. & de Roever, W.-P. (2005). 29 New Unclarities in the Semantics of UML 2.0 State Machines. In *Formal Methods and Software Engineering* (pp. 52–65). Springer Berlin Heidelberg.

Feiler, P. H. (2010). Model-based validation of safety-critical embedded systems. *2010 IEEE Aerospace Conference*, pp. 1–10.

Feiler, P. H., Gluch, D. P. & Hudak, J. J. (2006). *The Architecture Analysis & Design Language (AADL): An Introduction*. Software Engineering Institute, Carnegie Mellon University.

Feiler, P. H., Delange, J. & Wrage, L. (2016). *A Requirement Specification Language for AADL.* Software Engineering Institute, Carnegie Mellon University.

Ferrari, A., Fantechi, A., Magnani, G., Grasso, D. & Tempestini, M. (2013). The Metrô Rio case study. *Science of Computer Programming*, 78(7), 828–842.

Fifarek, A. W., Wagner, L. G., Hoffman, J. A., Rodes, B. D., Aiello, M. A. & Davis, J. A. (2017). SpeAR v2.0: Formalized Past LTL Specification and Analysis of Requirements. In *Proceedings of NASA Formal Methods: 9th International Symposium (NFM)* (pp. 420–426).

Finkelstein, A., Gabbay, D., Hunter, A., Kramer, J. & Nuseibeh, B. (1994). Inconsistency handling in multiperspective specifications. *IEEE Transactions on Software Engineering*, 20(8), 569–578.

Finlayson, M. A. (2014). Java libraries for accessing the Princeton WordNet: Comparison and evaluation. *In Proceedings of the 7th Global Wordnet Conference*.

Gaucher, F. & Génevaux, Y. (2017). Debugging Embedded Systems Requirements Before The Design Begins: "The Beginning is the Most Important Part of the Work" - Plato. *Ada Letters*, 36(2), 58–59.

Graf, S., Ober, I. & Ober, I. (2006). A real-time profile for UML. *International Journal on Software Tools for Technology Transfer*, 1–15.

Hawkins, R., Habli, I., Kelly, T. & McDermid, J. (2013). Assurance cases and prescriptive software safety certification: A comparative study. *Safety Science*, 59, 55–71.

Heimdahl, M. P. E. (2007). Safety and Software Intensive Systems: Challenges Old and New. *Future of Software Engineering (FOSE)*, pp. 137–152.

Herrmannsdörfer, M. & Berenbach, B. (2008). Tabular Notations for State Machine-Based Specifications. *The Journal of Defense Software Engineering*, 18–23.

Hooman, J., Kugler, H., Ober, I., Votintseva, A. & Yushtein, Y. (2007). Supporting UML-based development of embedded systems by formal techniques. *Software & Systems Modeling*, 7(2), 131–155.

Horváth, Á., Ráth, I. & Starr, R. R. (2015). Massif - the love child of Matlab Simulink and Eclipse. *EclipseCon*. Consulted at https://www.eclipsecon.org/na2015/session/massif-love-child-matlab-simulink-and-eclipse.

Huang, P., Jiang, K., Guan, C. & Du, D. (2018). Towards Modeling Cyber-Physical Systems with SysML/MARTE/pCCSL. *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, 01, 264–269.

Huhn, M. & Hungar, H. (2007). 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)*, pp. 201–237.

IncQuery Labs. (2017). Massif: MATLAB Simulink Integration Framework for Eclipse. Consulted at https://incquerylabs.com/en/page/show/massif.

IST. (2001). Flight Control Voting and Monitoring Case Study. Consulted at http://www-omega.imag.fr/cs/IAI/IAI.php.

Kelly, S. & Tolvanen, J. (2007). *Domain-Specific Modeling: Enabling Full Code Generation*. IEEE.

Kolovos, D., Rose, L., García-Domínguez, A. & Paige, R. (2018). The Epsilon Comparison Language (ECL). In *The Epsilon Book*. Eclipse Epsilon Project.

Kuroki, Y., Yoo, M. & Yokoyama, T. (2016, 03). A Simulink to UML model transformation tool for embedded control software development. *2016 IEEE International Conference on Industry Technology (ICIT)*, pp. 700–706.

Kurtev, I., Bézivin, J., Jouault, F. & Valduriez, P. (2006). Model-based DSL Frameworks. *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, (OOPSLA '06), 602–616.

Le Sergent, T., Dormoy, F.-X. & Le Guennec, A. (2016, 01). Benefits of Model Based System Engineering for Avionics Systems. *Proceedings of the 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*.

Lee, E. A. (2010). Disciplined Heterogeneous Modeling. *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010)*, pp. 273–287.

Lempia, D. L. & Miller, S. P. (2009). *Requirements Engineering Management Handbook*. Washington, DC, USA: National Technical Information Service (NTIS).

Leveson, N. G., Heimdahl, M. P. E., Hildreth, H. & Reese, J. D. (1994). Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9), 684–707.

Levine, W. S. (1996). *The control handbook*. CRC Press New York.

Luo, Y., van den Brand, M., Engelen, L., Favaro, J., Klabbers, M. & Sartori, G. (2013). Extracting Models from ISO 26262 for Reusable Safety Assurance. In *Safe and Secure Software Reuse: 13th International Conference on Software Reuse (ICSR)* (pp. 192–207). Berlin, Heidelberg: Springer Berlin Heidelberg.

MathWorks. (2018a). Simulink - Simulation and Model-Based Design. Consulted at https://www.mathworks.com/products/simulink.html.

MathWorks. (2018b). Stateflow. Consulted at https://www.mathworks.com/products/stateflow.html.

MathWorks Automotive Advisory Board. (2012). *Control Algorithm Modeling Guidelines using MATLAB, Simulink, and Stateflow*.

McGregor, J. D., Gluch, D. P. & Feiler, P. H. (2017). Analysis and Design of Safety-critical, Cyber-Physical Systems. *Ada Letters*, 36(2), 31–38.

Metayer, N. (2018). *An assurance level sensitive UML profile for supporting DO-178C*. (Master's thesis, École de Technologie Supérieure).

Metayer, N., Paz, A. & El Boussaidi, G. (2019). Modelling DO-178C Assurance Needs: A Design Assurance Level-Sensitive DSL. *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 1–8.

Micouin, P. (2008). Toward a Property Based Requirements Theory: System Requirements Structured As a Semilattice. *Systems Engineering*, 11(3), 235–245.

Miller, G. A. (1995). WordNet: A Lexical Database for English. *Communications of the ACM*, 38(11), 39–41.

Mosterman, P. & Ghidella, J. (2018). Fault Detection Control Logic in an Aircraft Elevator Control System. Consulted at https://www.mathworks.com/help/stateflow/examples/fault-detection-control-logic-in-an-aircraft-elevator-control-system.html.

Moy, Y., Ledinot, E., Delseny, H., Wiels, V. & Monate, B. (2013). Testing or Formal Verification: DO-178C Alternatives and Industrial Experience. *IEEE Software*, 30(3), 50–57.

Nair, S., de la Vara, J. L., Sabetzadeh, M. & Briand, L. (2014). An extended systematic literature review on provision of evidence for safety certification. *Information and Software Technology*, 56(7), 689–717.

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. *Information and Software Technology*, 54(6), 569–590.

OMG. (2011). *UML profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems*. Consulted at http://www.omg.org/spec/MARTE.

OMG. (2013). *OMG Meta Object Facility (MOF) Core Specification*. Consulted at https://www.omg.org/spec/MOF.

OMG. (2017a). *Systems Modeling Language*. Consulted at http://www.omg.org/spec/SysML.

OMG. (2017b). *OMG Unified Modeling Language (OMG UML)*. Consulted at http://www.omg.org/spec/UML.

OMG. (2018). *Structured Assurance Case Metamodel (SACM)*. Consulted at http://www.omg.org/spec/SACM/2.0/.

OpenDo. (2011). Project P. Consulted at http://www.open-do.org/projects/p/.

Özçelik, O. & Altilar, D. T. (2015). Test-Driven Approach for Safety-Critical Software Development. *Journal of Software*, 10(7), 904–911.

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. *Information and Software Technology*, 55(5), 836 - 864.

Paz, A. & El Boussaidi, G. (2016). On the Exploration of Model-Based Support for DO-178C-Compliant Avionics Software Development and Certification. *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 229–236.

Paz, A. & El Boussaidi, G. (2017). *Landing Gear Control Software: An avionics software development case study*. Montreal, Canada: École de Technologie Supérieure. Consulted at http://dx.doi.org/10.13140/RG.2.2.34900.19848.

Paz, A. & El Boussaidi, G. (2018). Building a Software Requirements Specification and Design for an Avionics System: An Experience Report. *Proceedings of the 33rd ACM Symposium on Applied Computing (SAC)*, pp. 1262–1271.

Paz, A. & El Boussaidi, G. (2019a). Breesse: Live bridge for the Eclipse Modeling Framework ecosystem and the MathWorks Simulink and Stateflow ecosystem. Consulted at https://github.com/afpaz/breesse.

Paz, A. & El Boussaidi, G. (2019b). A Requirements Modelling Language to Facilitate Avionics Software Verification and Certification. *Proceedings of the 6th International Workshop on Requirements Engineering and Testing (RET)*, pp. 1–8.

Paz, A. & El Boussaidi, G. (2019c). Supporting Consistency in the Heterogeneous Design of Safety-Critical Software. *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, 1, 37–46.

Paz, A. & El Boussaidi, G. (2019d). *Breeze through Safety-Critical System Model-Based Design with EMF, Simulink and Stateflow*. Unpublished manuscript, École de Technologie Supérieure, Montreal, Canada.

Paz, A. & El Boussaidi, G. (2019e). SpecML: Requirements Specification Modelling Language. Consulted at https://github.com/afpaz/specml.

Paz, A. & El Boussaidi, G. (2019f). checsdm: Consistency of Heterogeneous Embedded Control System Design Models. Consulted at https://github.com/afpaz/checsdm.

Paz, A., El Boussaidi, G. & Mili, H. (2020). checsdm: A Method for Ensuring Consistency in Heterogeneous Safety-Critical System Design. *IEEE Transactions on Software Engineering*, 1–27. Accepted for publication.

Pettit, R. G., Mezcciani, N. & Fant, J. (2014). On the Needs and Challenges of Model-Based Engineering for Spaceflight Software Systems. *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pp. 25–31.

Potter, B. (2012). *Complying with DO-178C and DO-331 using Model-Based Design*. MathWorks.

Potter, B. (2016). DO-178 Case Study. Consulted at https://www.mathworks.com/matlabcentral/fileexchange/56056-do178_case_study.

Radio Technical Commission for Aeronautics. (2011a). *Software Considerations in Airborne Systems and Equipment Certification*. RTCA DO-178C.

Radio Technical Commission for Aeronautics. (2011b). *Model-Based Development and Verification Supplement to DO-178C and DO-278A*. RTCA DO-331.

Radio Technical Commission for Aeronautics. (2011c). *Object-Oriented Technologies and Related Techniques Supplement to DO-178C and DO-278A*. RTCA DO-332.

Roques, P. (2016, 01). MBSE with the ARCADIA Method and the Capella Tool. *Proceedings of the 8th European Congress on Embedded Real Time Software and Systems*, (ERTS 2016), 1–10.

Rozanski, N. & Woods, E. (2011). *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives* (ed. 2). Addison-Wesley Professional.

Ruiz, A., Juez, G., Espinoza, H., de la Vara, J. L. & Larrucea, X. (2016). Reuse of safety certification artefacts across standards and domains: A systematic approach. *Reliability Engineering & System Safety*, 158, 153–171.

Runeson, P. & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2), 131–164.

SAE. (2017). *Architecture Analysis & Design Language (AADL)*. SAE AS5506C.

Sakairi, T., Palachi, E., Cohen, C., Hatsutori, Y., Shimizu, J. & Miyashita, H. (2012). Designing a control system using SysML and Simulink. *Proceedings of the SICE Annual Conference*, pp. 2011–2017.

Sarkis, A. & Dias, L. A. V. (2014). A Set of Rules for Production of Design Models Compliant with Standards DO-178C and DO-331. *11th International Conference on Information Technology: New Generations*, pp. 27–32.

Schamai, W., Buffoni, L., Albarello, N., Fontes De Miranda, P. & Fritzson, P. (2015). An Aeronautic Case Study for Requirement Formalization and Automated Model Composition in Modelica. *Proceedings of the 11th International Modelica Conference*, (118), 911–920.

Schmidt, D. C. (2006). Model-Driven Engineering. *Computer*, 39(2), 25–31.

Selic, B. (2003). The pragmatics of model-driven development. *IEEE Software*, 20(5), 19–25.

Selic, B. (2007, 05). A Systematic Approach to Domain-Specific Language Design Using UML. *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pp. 2–9.

Sjöstedt, C.-J., Shi, J., Törngren, M., Servat, D., Chen, D.-J., Ahlsten, V. & Lönn, H. (2008). Mapping Simulink to UML in the design of embedded systems: Investigating scenarios and transformations. *Proceedings of the 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems*, (OMER 4), 137–160.

Spitzer, C. R. (2007). *Avionics: Elements, Software and Functions* (ed. 1). CRC Press.

Stallbaum, H. & Rzepka, M. (2010). Toward DO-178B-compliant Test Models. *2010 Workshop on Model-Driven Engineering, Verification, and Validation (MoDeVVa)*, pp. 25–30.

Sztipanovits, J. (2007). Composition of Cyber-Physical Systems. *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*, pp. 3–6.

Tanaka, K., Inaho, S., Hatano, M., Kuroki, Y., Yoo, M. & Yokoyama, T. (2017). An Extended Simulink to UML Model Transformation Tool for Embedded Control Software Development. *Proceedings of the 2017 International Conference on Industrial Design Engineering (ICIDE)*, pp. 76–81.

The Eclipse Foundation. (2017a). EMF Documentation. Consulted at https://www.eclipse.org/modeling/emf/docs/.

The Eclipse Foundation. (2017b). Papyrus Modeling Environment. Consulted at https://www.eclipse.org/papyrus/.

The Eclipse Foundation. (2017c). Viatra. Consulted at https://www.eclipse.org/viatra/.

van den Brand, M. & Groote, J. F. (2015). Software engineering: Redundancy is key. *Science of Computer Programming*, 97, 75–81.

Vangheluwe, H., De Lara, J. & Mosterman, P. J. (2002). An introduction to multi-paradigm modelling and simulation. *Proceedings of the AIS'2002 conference (AI, Simulation and Planning in High Autonomy Systems)*, pp. 9–20.

Varró, D. (2016). Incremental Queries and Transformations: From Concepts to Industrial Applications. *SOFSEM 2016: Theory and Practice of Computer Science*, pp. 51–59.

Whalen, M. W. (2000). *A formal semantics for the Requirements State Machine Language Without Events (RSML-e)*. (Master's thesis).

White, J. & Reza, H. (2012, 11). Deriving DO-178C Requirements Within the Appropriate Level of Hierarchy. *The 7th International Conference on Software Engineering Advances (ICSEA)*, pp. 430–435.

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.

Yin, R. K. (2008). *Case Study Research: Design and Methods (Applied Social Research Methods)* (ed. 4). Sage Publications.

Yu, H., Ma, Y., Glouche, Y., Talpin, J.-P., Besnard, L., Gautier, T., Guernic, P. L., Toom, A. & Laurent, O. (2011). System-level Co-simulation of Integrated Avionics Using Polychrony. *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC)*, pp. 354–359.

Zoughbi, G., Briand, L. & Labiche, Y. (2011). Modeling Safety and Airworthiness (RTCA DO-178B) Information: Conceptual Model and UML Profile. *Software & Systems Modeling*, 10(3), 337–367.