POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Towards Reliable and Efficient Safety-Critical Systems: A DO-178C Framework for UAV Development

HENRIQUE AMARAL MISSON
Département de génie informatique et génie logiciel

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Génie informatique

Mai 2025

© Henrique Amaral Misson, 2025.

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cette thèse intitulée :

Towards Reliable and Efficient Safety-Critical Systems: A DO-178C Framework for UAV Development

présentée par **Henrique AMARAL MISSON**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

Tarek OULD BACHIR, président
Gabriela NICOLESCU, membre et directrice de recherche
Felipe GÖHRING DE MAGALHÃES, membre et codirecteur de recherche
John MULLINS, membre et codirecteur de recherche
Lina MARSSO, membre
Fabiano HESSEL, membre externe

DEDICATION

To my dear wife,
for her support, dedication and for never letting me give up, you are my greatest inspiration.
To my parents,
for always believing in me and encouraging me to fly higher.
To my sisters,
for always standing by me.
To my niece Sofia,
for bringing happiness and new purpose to our family.
To my grandmother,
for always encouraging me and never letting me give up.
To everyone in my family and friends,
for all the moments and words of encouragement.

ACKNOWLEDGEMENTS

This journey would not have been possible without the guidance, support, and encouragement of many individuals, to whom I am deeply grateful.

First and foremost, I would like to sincerely thank my supervisor, Professor Gabriela Nicolescu, for her consistent support, commitment, and invaluable lessons. Her support was crucial, particularly in the early stages of this journey, and her perspectives have greatly influenced the direction of this research.

I am truly thankful to Felipe Göhring de Magalhães for his constant support. Without his encouragement and assistance, this journey would have been considerably more difficult. I am so grateful to him because his presence changed everything.

I want to express my sincere gratitude to Professor John Mullins, whose influence was crucial in leading me to Polytechnique Montréal to follow this aspiration. His encouragement and faith in my abilities have been crucial in enabling this accomplishment.

I thank the members of the jury, Fabiano Hessel, Lina Marsso and Tarek Ould Bachir, for their willingness to assess this thesis.

I wish to express my gratitude and heartfelt thanks to the sponsors of this research, the Mitacs program and Humanitas Solutions, for their financial support. A special thanks to Maroua Ben Attia and Abdo Shabah for their involvement and contributions to this work.

Finally, I would like to express my appreciation to all my friends at the HESL laboratory for their companionship and support. I would especially like to thank Rim, who has contributed and fought with me during this journey.

Thank you to everyone who has contributed to this path and helped make this accomplishment possible.

RÉSUMÉ

Des progrès considérables ont été réalisés dans le développement des véhicules aériens sans pilote (UAV), qui permettent désormais des opérations autonomes sophistiquées dans divers domaines, tels que la mobilité aérienne urbaine, la logistique et la surveillance. Ces véhicules doivent interagir avec d'autres systèmes avioniques en opérant dans des environnements hautement dynamiques et critiques en termes de sûreté, tout en restant fiables et en respectant des réglementations aéronautiques strictes. Leur capacité à opérer au-delà de la ligne de visée (BVLOS) et dans un espace aérien partagé souligne l'importance cruciale de la sûreté. Conformément aux meilleures pratiques, le développement de logiciels pour drones doit être conforme aux normes industrielles telles que DO-178C, qui définit les lignes directrices pour la certification des logiciels dans les systèmes aéroportés. Le respect de ces normes exige une vérification, une validation et une traçabilité approfondies, toutes essentielles pour prouver l'exactitude et la robustesse du logiciel.

Bien que la conformité à la norme DO-178C présente de nombreux avantages, la réalisation de ses objectifs s'accompagne de défis considérables. Assurer la traçabilité des exigences de haut niveau (HLR) à la mise en œuvre, générer les artefacts de certification nécessaires et maintenir la cohérence tout au long du cycle de développement du logiciel exige un effort manuel important et des connaissances spécialisées. En outre, il devient plus difficile de valider la correction fonctionnelle, les propriétés de sûreté et la robustesse contre les défaillances en raison de la complexité des logiciels modernes pour drones, en particulier dans les systèmes autonomes. Les méthodologies de développement conventionnelles dépendent souvent de procédures manuelles et de techniques informelles, ce qui rend difficile la conformité avec les objectifs de la DO-178C tout en gérant efficacement les tâches de vérification et de validation. La résolution de ces problèmes nécessite des approches structurées qui combinent les méthodes formelles et l'automatisation afin d'optimiser le processus de développement et d'améliorer la fiabilité des logiciels critiques.

Pour relever ces défis, cette recherche présente une méthodologie conforme à la norme DO-178C qui intègre des méthodes formelles avec des outils automatisés pour faciliter le développement et la certification de logiciels pour drones. Une deuxième contribution est l'introduction d'un langage de modélisation spécifique au domaine (DSML) conçu pour définir des exigences de haut niveau en mettant l'accent sur les aspects liés à la sûreté. Ce DSML assure la conformité avec les objectifs de la DO-178C et permet une modélisation précise des exigences. Un cadre de transformation des modèles est également créé pour générer automatiquement

une architecture logicielle à partir d'exigences de haut niveau, minimisant ainsi la dépendance à l'égard des transitions manuelles et améliorant la traçabilité. En s'appuyant sur des techniques basées sur des modèles et sur l'automatisation, l'approche proposée améliore la fiabilité des logiciels tout en réduisant les efforts nécessaires à la certification.

La méthodologie est validée à l'aide d'une étude de cas portant sur un système d'évitement des collisions de drones, un composant critique conçu pour détecter et atténuer le risque de collisions en vol. L'étude de cas illustre comment le cadre de transformation de modèle permet la dérivation automatique de l'architecture logicielle et comment le DSML proposé soutient la spécification formelle des exigences de sûreté. L'efficacité de l'approche est évaluée en examinant dans quelle mesure elle améliore la cohérence, minimise les erreurs et produit des artefacts de certification conformes à la norme DO-178C. Les résultats démontrent les avantages de la combinaison de l'automatisation et de la modélisation formelle lors du développement de logiciels critiques de drones.

En conclusion, cette recherche contribue à faire progresser les pratiques d'ingénierie logicielle pour les systèmes aéroportés en introduisant une approche structurée et automatisée pour la spécification des exigences de haut niveau et la génération d'architecture. La méthodologie proposée améliore la traçabilité, réduit la complexité du développement et aide à répondre aux exigences de certification en comblant le fossé entre les exigences et l'architecture logicielle. Les résultats de cette recherche offrent des perspectives significatives pour améliorer l'efficacité et la rigueur du développement de logiciels dans le domaine des drones, ce qui permettra de créer des systèmes autonomes plus fiables et certifiables.

Mots-clés: DO-178C, Systèmes de sûreté critiques, Développement de logiciels, Ingénierie basée sur les modèles, Langage de modélisation spécifique à un domaine, Transformation de modèles.

ABSTRACT

Significant progress has been made in developing Unmanned Aerial Vehicles (UAVs), which now allows for sophisticated autonomous operations in various fields, such as urban air mobility, logistics, and surveillance. These vehicles must interact with other avionics systems while operating in highly dynamic and safety-critical environments while maintaining dependability and adhering to strict aviation regulations. Their ability to operate beyond visual line-of-sight (BVLOS) and within shared airspace emphasizes the critical importance of safety. In line with best practices, UAV software development should comply with industry standards like DO-178C, which sets forth guidelines for certifying software in airborne systems. Meeting these standards requires thorough verification, validation, and traceability, all essential to prove the software's correctness and robustness.

Even though DO-178C compliance has many advantages, satisfying its objectives comes with considerable challenges. Ensuring the traceability from high-level requirements (HLRs) to implementation, generating necessary certification artifacts, and maintaining consistency throughout the software development lifecycle demands substantial manual effort and specialized knowledge. Furthermore, it becomes more challenging to validate functional correctness, safety properties, and robustness against failures due to the complexity of modern UAV software, especially in autonomous systems. Conventional development methodologies frequently depend on manual procedures and informal techniques, which makes it difficult to ensure compliance with DO-178C objectives while effectively handling verification and validation tasks. Tackling these issues necessitates organized approaches that combine formal methods and automation to optimize the development process and improve the dependability of safety-critical software.

To address these challenges, this research presents a DO-178C-compliant methodology that integrates formal methods with automated tools to facilitate the development and certification of UAV software. Another contribution is introducing a domain-specific modelling language (DSML) designed to define high-level requirements with an emphasis on safety-related aspects. This DSML guarantees conformity with DO-178C objectives and permits accurate requirement modelling. A framework for model transformation is also created to automatically produce software architecture from high-level requirements, minimizing the dependence on manual transitions and improving traceability. By leveraging model-based techniques and automation, the proposed approach enhances software reliability while reducing the effort required for certification.

The methodology is validated using a case study of a UAV collision avoidance system, a critical component designed to detect and mitigate the risk of mid-air collisions. The case study illustrates how the model transformation framework enables the automatic derivation of software architecture and how the suggested DSML supports the formal specification of safety requirements. The approach's effectiveness is assessed by examining how well it enhances consistency, minimizes errors, and produces certification artifacts in accordance with DO-178C. The findings demonstrate the advantages of combining automation and formal modelling when developing safety-critical UAV software.

In conclusion, this research contributes to advancing software engineering practices for airborne systems by introducing a structured and automated approach for high-level requirement specification and architecture generation. The proposed methodology improves traceability, reduces development complexity, and helps meet certification requirements by bridging the gap between requirements and software architecture. The results of this research offer significant insights into enhancing the effectiveness and thoroughness of software development in the UAV domain, leading to more dependable and certifiable autonomous systems.

Keywords: DO-178C, Safety-critical systems, Software development, Model-based engineering, Domain-specific modelling language, Model transformation.

TABLE OF CONTENTS

DEDICATION iii

ACKNOWLEDGEMENTS iv

RÉSUMÉ v

ABSTRACT vii

TABLE OF CONTENTS ix

LIST OF TABLES xii

LIST OF FIGURES xiii

LIST OF SYMBOLS AND ACRONYMS xvii

LIST OF APPENDICES xix

CHAPTER 1 INTRODUCTION 1

1.1 Problem Definition 2

1.2 Research Objectives 3

1.3 Research Contributions 5

1.4 Thesis Outline 6

1.5 List of Publications 7

CHAPTER 2 BACKGROUND AND LITERATURE REVIEW 9

2.1 Avionics Software Standards 9

2.1.1 DO-178C 9

2.1.2 ARINC653 13

2.2 Formal Methods 14

2.3 Model-Based Engineering 15

2.3.1 Domain-Specific modelling Language 16

2.3.2 Model Transformation 18

2.4 Tools and Languages 19

2.4.1 AADL 19

2.4.2 DO-178C software development tools 21

2.4.3 Formal Methods Tooling 25
2.5 Related Works 29
2.5.1 Methodologies for Software Development and Certification in Avionics 29
2.5.2 Approaches for Specifying Software Requirements in Safety-Critical Systems 35
2.5.3 Transitioning from Requirements to Software Architecture 38
2.6 Chapter Summary 41

CHAPTER 3 RESEARCH METHODOLOGY 42

CHAPTER 4 A METHODOLOGY FOR DO-178C-COMPLIANT SOFTWARE DEVELOPMENT: INTEGRATING FORMAL METHODS AND AUTOMATED TOOLS 45
4.1 Chapter Overview 45
4.2 Methodology 46
4.2.1 Proposed Methodology 46
4.2.2 Formal Methods and Automated Tools: A Unified Approach 48
4.3 Process Description 49
4.3.1 Software Requirements Phase 49
4.3.2 Software Design Phase 51
4.3.3 Software Coding Phase 55
4.3.4 Software Integration Phase 58
4.4 Case Study: Collision Avoidance System for UAVs 60
4.4.1 Overview of Collision Avoidance System 60
4.4.2 Implementation of Methodology 62
4.4.3 Data Collection and Analysis 81
4.5 Results and Discussion 85
4.5.1 Evaluation Against DO-178C Objectives 85
4.5.2 Impact of Methodology 86
4.5.3 Case Study Insights 88
4.5.4 Practical Implications: Best Practices for DO-178C Compliance . . . 90
4.6 Chapter Summary 93

CHAPTER 5 REDAML: A MODELLING LANGUAGE FOR DO-178C HIGH-LEVEL REQUIREMENTS IN AIRSPACE SYSTEMS 95
5.1 Chapter Overview 95
5.2 ReDaML 96
5.2.1 Abstract syntax 96

5.2.2 Concrete syntax 100
5.2.3 Semantics 100
5.3 Application Scenario 102
5.3.1 Collision Avoidance System Overview 102
5.3.2 Modelling with ReDaML 102
5.4 Considerations 104
5.5 Chapter Summary 105

CHAPTER 6 BRIDGING THE GAP: A DO-178C COMPLIANT FRAMEWORK FOR REQUIREMENTS-TO-ARCHITECTURE TRANSITION 107
6.1 Chapter Overview 107
6.2 Framework Overview: Bridging Requirements to Architecture 108
6.2.1 Methodology 109
6.2.2 Model Transformation 110
6.3 Case Study 120
6.4 Discussion & Evaluation 125
6.5 Chapter Summary 127

CHAPTER 7 CONCLUSION 128
7.1 Summary of Works 128
7.2 Limitations 130
7.3 Future Research 131

REFERENCES 133

APPENDICES 142

LIST OF TABLES

Table 4.1 Summary of Activities, Outputs, and DO-178C Objectives Addressed in the Software Requirements Phase 52
Table 4.2 Summary of Activities, Outputs, and DO-178C Objectives Addressed in the Software Design Phase 56
Table 4.3 Summary of Verification and Traceability Activities, Tools, Outputs, and DO-178C Objectives Addressed 94
Table 5.1 Requirements for ReDaML. 97
Table 5.2 Semantic validation rules using OCL constraints. 101
Table 6.1 Details of AADL elements for representing processor, memory and communication resources in template model. 114
Table 6.2 Transformation rule *Create_External_App* with pre and post conditions 117
Table 6.3 Test cases to validate the correctness of transformation rules. Each scenario was created from an empty initial scenario. 119
Table 6.4 Summary of DO-178C objectives satisfied through the framework . . 126
Table B.1 Validation rules with corresponding expressions and problem levels . 169
Table Table B.2 High-Level Requirements (HLRs) for the Collision Avoidance System 174

LIST OF FIGURES

Figure 2.1 Overview of DO-178C processes 10
Figure 2.2 Overview and an example of DO-178C processes and their verification outputs objectives 12
Figure 2.3 Schematic view of model checking technique (*Source*: [12]). 15
Figure 2.4 Overview of the DSML (Source: [24]) 17
Figure 2.5 AADL summary elements. Source: [29] 20
Figure 2.6 TBmanager view showing the traceability between different requirement levels and source code. Extracted from `https://ldra.com/products/tbmanager` 23
Figure 2.7 TBvision view showing code review violations. Extracted from [35] . 24
Figure 3.1 Research Methodology 43
Figure 4.1 Expanded view of the software requirements process. 50
Figure 4.2 Expanded view of the software design process. 53
Figure 4.3 Expanded view of the software coding process. 57
Figure 4.4 Expanded view of the software integration process. 60
Figure 4.5 TBmanager relationship view between SRATS and HLR 63
Figure 4.6 Results for checking the InadequateCollisionThreatHandling assertion after refinement 65
Figure 4.7 Traceability matrix between SRATS and HLRs 65
Figure 4.8 Checklist used for peer review of HLRs during the validation 66
Figure 4.9 DO-178C objectives for verification of outputs of software requirement process in TBmanager 67
Figure 4.10 AADL model of the traffic detection subsystem 68
Figure 4.11 Checklist used for peer review of the architecture 69
Figure 4.12 Results for checking the HLR_001_TrafficDetection assertion 71
Figure 4.13 Traceability matrix between HLRs and LLRs 72
Figure 4.14 FTA for the Traffic Detection subsystem 73
Figure 4.15 DO-178C objectives for verification of outputs of the software design process in TBmanager 74
Figure 4.16 Simulink model of the CAS showing subsystems and data flow. 75
Figure 4.17 TBmanager relationship view showing the traceability between three-requirement levels (SRATS, HLR and LLR) and the source code functions 76
Figure 4.18 Code review report generated by LDRA 77

Figure 4.19 Code quality report generated by LDRA 78
Figure 4.20 Code coverage analysis report generated by LDRA 79
Figure 4.21 Bounded Model Checking results using ESBMC for CAS code 80
Figure 4.22 Tbrun Unit / Module Test report 81
Figure 4.23 DO-178C objectives for verification of outputs of the software coding process in TBmanager 82
Figure 4.24 Traceability matrix between LLRs and Code 84
Figure 4.25 LDRA TBmanager project coverage summary for CAS requirements and test cases 84
Figure 4.26 Defect summary report 85
Figure 5.1 Simplified metamodel of ReDaML [91]. It shows the elements and their relationships of an HLR specification for critical systems based on the DO-178C standard. 98
Figure 5.2 Modelling tool palette. 103
Figure 5.3 Warning message from R1 rule. 104
Figure 5.4 HLR-4 and DHLR-4.1 modelled with ReDaML [91] diagram view. It depicts the HLR aspects according to DO-178C. The functional type HLR is traced to SRATS-4 on the right. Additionally, a derived requirement of the safety type related to the HLR is represented, together with a rationale behind it. Given that it is a safety type, ReDaML imposes the description to provide more details. 105
Figure 6.1 Workflow of the proposed Framework within the dotted rectangle and its relationship with DO-178C processes. It highlights the HLR modelling and verification, in which its model serves as the input of the model transformation to generate the architecture. 109
Figure 6.2 Structure of the model transformation process. 111
Figure 6.3 AADL components representing network interface containing the bus (in the left), processor with one partition (in the center), and memory with two segments (in the right) 113

Figure 6.4 Extraction of ReDaML metamodel [91]. It considers elements that model HLRs and their relationships with other components through interfaces. For example, a requirement specifying that the application must send or receive data from another application or piece of equipment. When this interaction is through sampling ports, i.e. the data must be fresh, the "*InterfaceData*" element is modelled. If it is an occasional interaction, via a queuing port, the element representing "*InterfaceEvent*" is modelled. 116
Figure 6.5 Example of HLR with interface description using ReDaML 118
Figure 6.6 Activity diagram illustrating the process of transforming High-Level Requirements (HLRs) into an Architecture model in AADL. 121
Figure 6.7 User interface for diagram validation in ReDaML. The figure displays HLR-4 and DHLR-4.1 models, as illustrated in Figure 5.4, alongside a context menu with the "Validate diagram" option. Validation results are displayed in the bottom left window for models with errors, while the bottom right window confirms an error-free model with no issues reported. 123
Figure 6.8 Transformation process in Eclipse environment. A menu is directly in the Eclipse tool where the HLR was modelled to facilitate the transformation. The image below represents the generated AADL files that form the architecture. 124
Figure 6.9 System view of architecture generated in AADL 125
Figure A.1 AADL model of the traffic detection subsystem 154
Figure A.2 AADL model of the traffic tracking subsystem 154
Figure A.3 AADL model of the collision evaluation subsystem 155
Figure A.4 AADL model of the threat prioritization subsystem 155
Figure A.5 AADL model of the maneuver determination subsystem 156
Figure A.6 AADL model of the maneuver command subsystem 156
Figure B.1 The complete ReDaML meta-model 162
Figure B.2 The complete ReDaML meta-model 163
Figure B.3 The complete ReDaML meta-model 164
Figure B.4 The complete ReDaML meta-model 164
Figure B.5 Concrete syntax specification model - diagram view 165
Figure B.6 Concrete syntax specification model - table view 166
Figure B.7 Concrete syntax specification model - tree view 166
Figure B.8 ReDaML's semantics view in Obeo 167

Figure B.9 HLR-1 and its DHLR modelled with ReDaML - diagram view 175
Figure B.10 HLR-2 and its DHLR modelled with ReDaML - diagram view 175
Figure B.11 HLR-3 and its DHLR modelled with ReDaML - diagram view 176
Figure B.12 HLR-4 and its DHLR modelled with ReDaML - diagram view 176
Figure B.13 HLR-5 and its DHLR modelled with ReDaML - diagram view 177
Figure B.14 HLR-6 and its DHLR modelled with ReDaML - diagram view 177
Figure B.15 HLR-8, HLR-9, HLR-10 and HLR-11 modelled with ReDaML - diagram view 178
Figure B.16 HLRs and its DHLRs modelled with ReDaML - table view 179
Figure B.17 Safety-related HLRs modelled with ReDaML - tree view 180

LIST OF SYMBOLS AND ACRONYMS

AADL	Architecture Analysis & Design Language
ACPS	Airspace Cyber-Physical System
ARINC653	Avionics Application Standard Software Interface
AST	Assisted Transformation of Models
BVLOS	Beyond Visual Line of Sight
CAS	Collision Avoidance Systems
DAL	Design Assurance Level
DHLR	Derived High-Level Requirements
DLLR	Derived Low-Level Requirements
DO-178C	Software Considerations in Airborne Systems and Equipment Certification
DO-331	Model-Based Development and Verification Supplement to DO-178C and DO-278A
DO-332	Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A
DO-333	Formal Methods Supplement to DO-178C and DO-278A
DSML	Domain-Specific Modelling Language
ESBMC	Efficient SMT-Based Bounded Model Checker
EVTOL	Electric Vertical Take-Off and Landing
FMEA	Failure Mode and Effects Analysis
FTA	Fault Tree Analysis
HLR	High-Level Requirement
IDE	Integrated Development Environment
IMA	Integrated Modular Avionics
IPC	Inter-Partition Communication
MBE	Model-Based Engineering
MBD	Model-Based Development
MBT	Model-Based Testing
MC/DC	Modified Condition/Decision Coverage
MOF	Meta-Object Facility
M2M	Model-To-Model
M2T	Model-To-Text
OCL	Object Constraint Language

RDAL	Requirements Definition and Analysis Language
REDAML	REquirements DAta Modelling Language
RSML	Requirements State Machine Language
RTCA	Radio Technical Commission for Aeronautics
RTOS	Real-Time Operating Systems
SAS	Software Accomplishment Summaries
SCE	Safety Context Element
SCS	Safety-Critical Systems
SRATS	System Requirements Allocated to Software
SRDM	Software Requirement Data Model
SRTM	Software Requirements Traceability Matrix
UAS	Unmanned Aerial Systems
UAV	Unmanned Aerial Vehicle
UML	Unified Modelling Language

LIST OF APPENDICES

Appendix A Collision Avoidance Software Development - Requirements and Design 142
Appendix B ReDaML: Development and Case Study 162

CHAPTER 1 INTRODUCTION

The rapid adoption of Unmanned Aerial Vehicles (UAVs) across diverse industries has underscored the critical importance of developing reliable and safe software to ensure their operation. UAVs, as safety-critical systems, face unique risks where software failures could lead to catastrophic consequences, including loss of life and property [1]. This has driven the demand for rigorous development methodologies that adhere to high certification standards [2], such as DO-178C [3], the industry-recognized regulatory framework for airborne software.

Developing software for UAVs presents unique challenges due to their safety-critical nature and operational complexity. UAVs must perform reliably in sensitive and dynamic environments, such as urban areas, vital infrastructure zones, and shared airspace with manned aircraft. Autonomous and semi-autonomous operations further compound their complexity, necessitating a systematic development approach to detect and mitigate potential hazards. In this context, the role of software is paramount in ensuring operational safety and adherence to stringent aviation regulations [4].

Strict standards like DO-178C play a critical role in reducing the risks associated with safety-critical systems by outlining a structured development lifecycle that emphasizes safety and reliability [5]. Certification requires a disciplined process, including thorough requirements development, robust architecture design, and systematic verification. These practices not only ensure compliance with regulations but also facilitate the seamless integration of UAVs into airspaces shared with manned aircraft [6].

A successful development process for safety-critical systems relies on several key factors, including well-documented system architecture and requirements, robust safety practices at every stage, and rigorous testing at all levels [7]. Transparency and traceability in requirements and design are essential for maintaining consistency and reducing errors throughout the lifecycle. Moreover, adherence to established standards and best practices strengthens the software's resilience, ensuring it can withstand the scrutiny of certification audits.

Technological advancements have introduced innovative approaches to address the challenges of developing safety-critical software. Formal methods, automated verification tools, and Model-Based Engineering (MBE) are increasingly utilized to manage the complexities of modern systems [8]. These methodologies enhance consistency, reduce human error, and enable early identification of potential issues, thus improving the overall reliability and safety of the software.

As technology evolves, software development must adapt to meet new challenges and opportunities. By leveraging advanced tools and methodologies, developers can effectively manage the increasing complexity of UAV systems while maintaining a strong commitment to safety and dependability [9]. This evolution not only ensures regulatory compliance but also paves the way for safer, more reliable software, supporting the growing role of UAVs in critical applications.

1.1 Problem Definition

DO-178C standards guide the software development process in avionics by directing rigorous lifecycle practices, such as the widely used V-model [10], which emphasizes verification and validation at every stage of development. This lifecycle begins with defining comprehensive requirements, ensuring consistency, clarity, and traceability. High-quality requirements are the foundation of successful software, influencing every subsequent phase, including design, implementation, and testing.

While standards provide a structured framework and best practices for guiding the software development lifecycle, strictly adhering to every step is not always guaranteed in practice. Companies often face unique constraints, such as limited resources, tight schedules, or evolving project requirements, which can challenge their ability to comply fully with all planning development activities. These constraints may lead to deviations or adaptations in the implementation of standards, adding further complexity to developing safety-critical software. Balancing compliance with practical reality necessitates careful prioritizing and inventive solutions to ensure that safety and reliability are not jeopardized, even in the face of such obstacles.

Designing a solid software development process with well-defined tasks is vital for addressing all critical areas while minimizing inefficiencies and rework. A well-defined procedure helps prevent the absence of crucial safety and dependability factors, which are especially important in safety-critical systems, such as UAVs. Effective testing at each stage of development is critical for identifying potential problems early on and ensuring that the program works reliably in every scenario. Formal methods, including theorem proving [11] and model checking [12], provide powerful tools for verifying system properties and boosting confidence in the software's correctness. However, implementing such processes is challenging due to their complexity, resource requirements, and the need to adhere to strict industry standards like DO-178C, which demands rigorous planning, implementation, and validation of every development phase.

One of the development processes is the requirements. High-level requirements (HLRs) form the software specification at the highest level, describing what is to be produced. They must be meticulously generated from system-level needs to guarantee completeness, clarity, and coverage of all crucial features, which makes this process extremely difficult. HLRs should be easy to use and understand, and they should be comprehensive enough to guide further development activities. They must also capture all safety-critical characteristics, ensuring the software adequately addresses hazards and meets its intended safety objectives. Finding this balance necessitates a thorough comprehension of the operating environment of the system as well as the regulations governing safety-critical software, which makes the requirements process an essential but challenging task.

Following the definition and validation of the HLRs, the software architecture, a crucial part of the design process, must be designed. The software's structural basis is provided by the architecture, which also directs code implementation and guarantees that the system satisfies its functional and safety objectives. However, creating an effective architecture from HLRs is a difficult undertaking that primarily relies on the knowledge and experience of developers. It entails converting abstract specifications into a concrete structure while preserving traceability and guaranteeing adherence to safety regulations. This process is critical to achieving a design that is both robust and adaptable to the complexities of critical software systems, such as those in UAVs.

In summary, to ensure accuracy and consistency throughout the lifecycle of safety-critical systems, it is crucial to establish a robust approach to software development. New technology integration is essential for making it easier to specify requirements, especially regarding safety and guaranteeing adherence to standard objectives like those outlined in DO-178C. Furthermore, designing the system architecture in strict alignment with these requirements ensures comprehensive coverage, a solid foundation for implementation, and maintained traceability. Together, these components form the backbone of a rigorous development process, enhancing safety, reliability, and compliance with certification standards.

1.2 Research Objectives

The main objective of this research is to assist the software development process in avionics by proposing techniques and methodologies that guide the development toward achieving certification. This goal is achieved through tools and approaches that enforce essential aspects during the development process, ultimately enhancing the quality and reliability of the software while ensuring compliance with rigorous standards. To accomplish this, specific objectives were defined as follows:

- **OBJ 1 - Propose a new methodology for software development by integrating formal methods activities and automated tools at all levels.**
  Effective methodologies with well-defined activities significantly improve software quality, particularly in safety-critical domains like aviation. Verification and testing play a crucial role in ensuring robustness and reliability, and formal methods have proven effective in identifying bugs early in the development process. Early detection and correction of issues help avoid costly rework and delays. Additionally, automated tools streamline resource and time management while facilitating compliance with certification standards. Integrating formal methods and automation throughout the development lifecycle forms the basis of a methodology aimed at increasing software quality and meeting the stringent demands of certification standards.

- **OBJ 2 - Propose a development of new approach for high-level requirements specification based on the DO-178C standard and enhancing safety-related aspects.**
  The HLRs phase provides the foundation for all subsequent processes in software development. The development of a domain-specific modelling language tailored for HLRs aims to support the development of safety-critical systems in compliance with DO-178C guidelines. This language enables engineers to specify HLRs with a structured formalism incorporating safety-related aspects, ensuring that critical requirements are appropriately highlighted and enforced. This approach improves the quality of requirements specification, reduces errors, and facilitates detailed analyses during development. Furthermore, employing an MBE approach helps manage the inherent complexity of safety-critical software, offering a robust framework for aligning the requirements process with regulatory objectives.

- **OBJ 3 - Propose an approach for generating automated architecture from high-level requirements.**
  Transitioning from high-level requirements to a comprehensive software architecture is a critical and challenging step in the development lifecycle, particularly in safety-critical domains. This research emphasizes the importance of capturing functional and non-functional requirements in the architecture while maintaining traceability across artifacts. Neglecting these aspects risks introduces costly errors and failures. To address these challenges, the research proposes a model transformation technique that systematically translates HLRs into architectural artifacts. This automated approach ensures completeness and consistency, reduces human error, and provides a structured framework for integrating functional and non-functional requirements. By adhering

to DO-178C objectives, the approach aims to make architecture development more accessible and reliable, especially for engineers with varying levels of expertise, while maintaining the stringent safety and reliability standards required for certification.

1.3 Research Contributions

In order to address the main challenges in developing UAV software for certification, as described in the sections above, three main questions were proposed to guide the development of this research.

- **RQ1**: How can formal methods and automated tools be effectively integrated into the software development process to facilitate compliance with DO-178C objectives?
- **RQ2**: How can a domain-specific modelling language improve the specification, validation, and traceability of High-Level Requirements, especially for safety-related requirements in avionics systems?
- **RQ3**: How can model transformation techniques be leveraged to automatically generate software architecture from software requirements while ensuring traceability and correctness?

Thesis Contributions Based on the research objectives and questions stated above, this thesis makes three key contributions to the field of avionics software development:

- **Contribution 1 - A DO-178C-Compliant Methodology with Formal Methods and Automation**: This research proposes a structured methodology that integrates formal methods into the software development lifecycle, aligning with DO-178C objectives. The methodology incorporates automated tools to streamline compliance verification and certification evidence generation, reducing manual efforts and improving the overall reliability of the development process.
- **Contribution 2 - A Domain-Specific Modeling Language for High-Level Requirements**: To enhance the specification and validation of safety-critical software, a domain-specific modelling language (DSML) is proposed to define HLRs in accordance with DO-178C. This language enforces consistency, facilitates traceability, and provides a structured way to capture safety-related requirements, ensuring alignment with regulatory standards.

- **Contribution 3 - A Framework for Automated Software Architecture Generation**: The thesis introduces a model transformation framework that automatically derives software architecture from HLRs. This framework ensures that the generated architecture adheres to the system's functional and safety requirements while maintaining traceability between requirements and design. By automating this process, the framework enhances development efficiency and reduces the risk of inconsistencies between requirements and implementation.

1.4 Thesis Outline

This thesis is organized into six chapters. In this chapter, an introduction with the research context is presented, along with the objectives and contributions of the thesis. Additionally, a list of publications produced during this research is provided. The next chapters are organized as follows:

- **Chapter 2: Background and Related Work**: This chapter introduces the key concepts necessary for understanding the research, including avionics standards such as DO-178C and ARINC 653, MBE, domain-specific modelling languages, and model transformation. Furthermore, it presents related work from the literature on topics connected to the thesis, such as methodologies for avionics software development, particularly in the context of UAVs, modelling languages for requirements specification in line with DO-178C, and model transformations for generating software architecture.

- **Chapter 3: Research Methodology**: This chapter details the research methodology in which the work structure is presented with stages followed and linking each contribution to the succeeding chapters.

- **Chapter 4: ReDaML – A Domain-Specific Modeling Language**: In this chapter, ReDaML, a domain-specific modelling language for specifying HLRs following DO-178C, is introduced. The language focuses on improving the capture of safety-related HLRs, enabling better analysis and facilitating effective communication among various disciplines involved in developing Airspace Cyber-Physical Systems (ACPS).

- **Chapter 5: Framework for Model Transformation**: This chapter presents the framework developed to support the generation of software architecture designs from HLRs modelled using ReDaML. The proposed model transformation approach ensures consistency and traceability, facilitating compliance with DO-178C objectives while reducing complexity in the architecture development process.

- **Chapter 6: Conclusion and Future Work**: The final chapter summarizes the main contributions and results of the research, highlighting its significance, limitations, and challenges encountered. Additionally, potential directions for future research are discussed.

1.5 List of Publications

Alongside this Ph.D., the following list of papers was published:

- Zrelli, R.*, Misson, H. A.*, Kamkuimo, S., Ben Attia, M., de Magalhães, F. G., & Nicolescu, G. (2025). Integrating Formal Methods and Automated Tools for DO-178C Compliance in UAV Software. Submitted to ACM Transactions on Embedded Computing Systems. (*: Equal Contribution)

- Misson, H. A., Zrelli, R., Ben Attia, M., Magalhaes, F. G., & Nicolescu, G. (2023, September). ReDaML: A Modeling Language for DO-178C High-Level Requirements in Airspace Systems. In Proceedings of the 34th International Workshop on Rapid System Prototyping (pp. 1-7).

- Misson, H.A., Zrelli, R., Ben Attia, M., Magalhaes, F.G.d., Shabah, A., & Nicolescu, G. (2025). Bridging the Gap: A DO-178C Compliant Framework for Requirements-to-Architecture Transition. Submitted to the Aerospace Systems journal.

Additionally, collaborations with other publications have been carried out.

- Kamkuimo, S. A., Magalhaes, F., Zrelli, R., Misson, H. A., Attia, M. B., & Nicolescu, G. (2023). Decomposition and Modeling of the Situational Awareness of Unmanned Aerial Vehicles for Advanced Air Mobility. Drones, 7(8), 501.

- Zrelli, R., Amaral Misson, H., Ben Attia, M., Gohring de Magalhães, F., Shabah, A., & Nicolescu, G. (2024, March). Natural2CTL: A Dataset for Natural Language Requirements and Their CTL Formal Equivalents. In International Working Conference on Requirements Engineering: Foundation for Software Quality (pp. 205-216). Cham: Springer Nature Switzerland.

- Zrelli, R., Misson, H.A., Ben Attia, M., Magalhaes, F.G.d., Shabah, A., & Nicolescu, G.: Advancing Formal Verification: Fine-tuning LLMs for Translating Natural Language requirements to CTL specifications. In: Proceedings of the 35th International Workshop on Rapid System Prototyping (2024). Accepted, to appear.

CHAPTER 2 BACKGROUND AND LITERATURE REVIEW

This chapter provides the background and context for comprehending the research contributions. It starts with an overview of crucial concepts, such as the DO-178C and ARINC 653 avionics standards, which set software development and partitioning guidelines in safety-critical systems. The chapter then delves into MBE, a development approach that utilizes models to tackle the complexities of modern systems. It also examines domain-specific modelling languages and model transformation techniques, which are central to this research.

Beyond the theoretical background, this chapter reviews related literature in areas relevant to the thesis. It explores methodologies for avionics software development, particularly for UAVs, emphasizing safety, reliability, and regulatory compliance. Additionally, the chapter examines existing modelling languages for requirements specification, focusing on their alignment with DO-178C objectives, and reviews prior research on model transformations for generating software architectures. By integrating theoretical insights with a survey of pertinent literature, this chapter establishes the foundation and identifies gaps addressed by this thesis.

2.1 Avionics Software Standards

2.1.1 DO-178C

RTCA DO-178C [3], titled "Software Considerations in Airborne Systems and Equipment Certification," is the de facto standard for developing software for airborne systems. This standard provides a comprehensive framework to ensure software safety, reliability, and compliance with airworthiness requirements, forming the foundation for regulatory certification processes. It is an objective-oriented guideline that defines the necessary activities and objectives to establish an acceptable level of confidence in software safety.

The standard introduces the concept of Design Assurance Levels (DALs) to categorize software based on its potential impact on system safety. DALs range from Level A, where failure can result in catastrophic consequences, to Level E, where failure does not affect safety. The level of assurance determines the number and rigour of objectives that must be satisfied. For instance, Level A requires compliance with 71 objectives, reflecting its critical safety implications. These objectives are fulfilled through activities structured across the software life cycle, which is divided into three phases, depicted in Figure 2.1. This figure highlights the main processes at the highest level that form the framework of the standard and their

relationships.

- Planning Phase: Defines the tasks, schedules, and plans for software development.
- Software Development Phase: Includes key processes such as requirements, design, coding, and integration.
- Integral Process: Runs concurrently with development and focuses on verification and validation, ensuring that the software meets both stakeholder expectations and certification objectives.

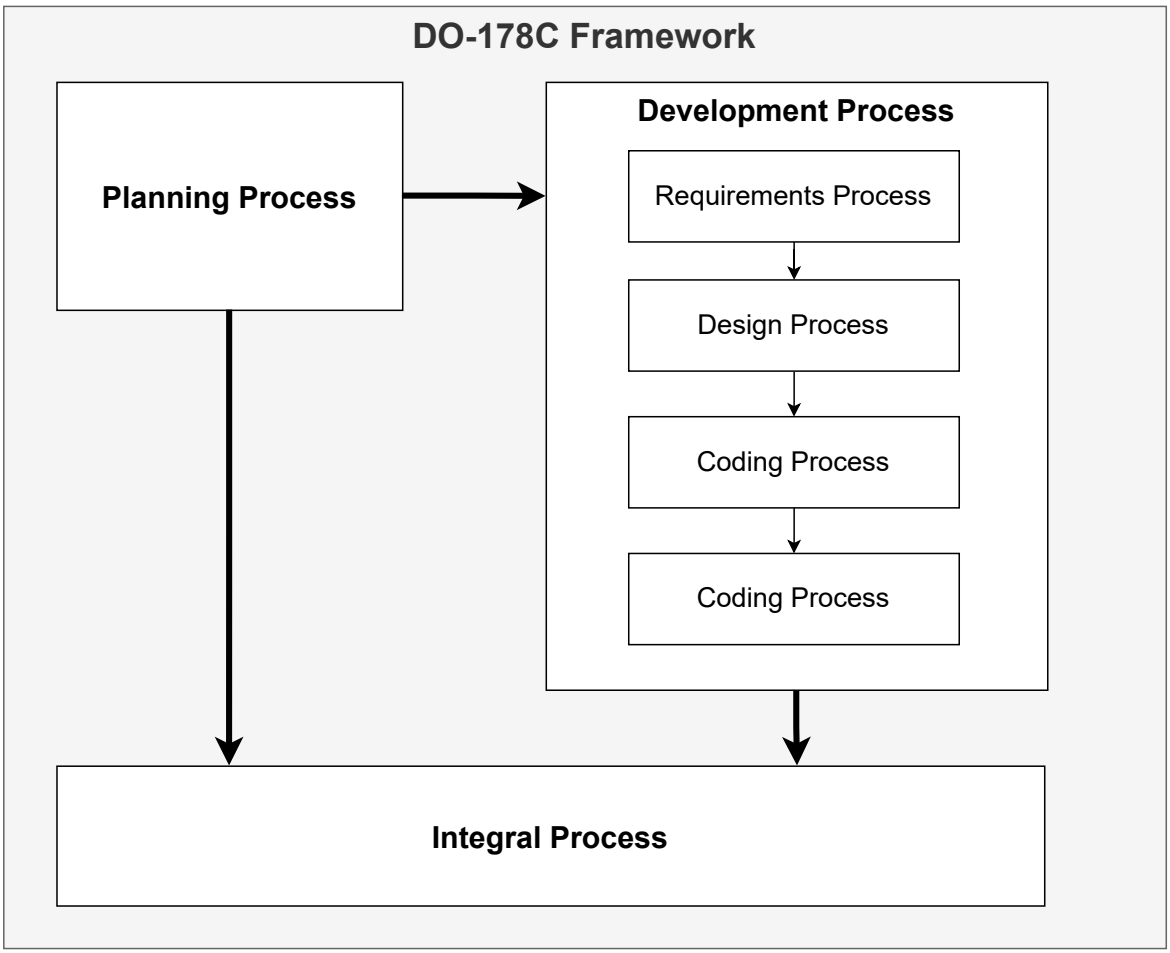

Figure 2.1 Overview of DO-178C processes

The current research focuses mainly on software development processes and proposes various integral process activities in the methodology developed. The requirements process within

DO-178C is crucial, as it defines Requirements Data derived from the System Requirements Allocated to Software (SRATS) and system architecture. These include HLRs and Derived High-Level Requirements (DHLRs), which refine HLRs to complement behaviours that are not directly traceable to SRATS. HLRs must be complete, consistent, and accurate, as they form the foundation for all subsequent activities.

The requirements data defined during the requirements process serve as the foundation for the subsequent design process, which focuses on creating the software architecture and defining the Low-Level Requirements (LLRs). These artifacts, collectively called Design Description in DO-178C, have unique but complementary functions. The architecture outlines the structure of the software, detailing the relationships between its various components. Meanwhile, the LLRs specify the precise functionalities to be implemented during the coding process, ensuring a seamless transition between design and implementation. In the coding process, developers write the source code based on the LLRs, transforming requirements into a functional software system. The integration process then takes the developed source code and compiles it into the executable object code, ensuring it can be correctly embedded in the target computer. This step also validates the correctness of the compiled software, preparing it for integration into the overall system. When combined, these procedures create a smooth transition from requirements to a software product that is both fully functional and certified.

Supplemental documents enhance DO-178C, such as DO-331 for model-based development and verification, and DO-333, which addresses the use of formal methods. DO-331 provides methodologies for building and validating models, emphasizes tool certification, and integrates model-based techniques into workflows, enabling efficient compliance with safety standards. DO-333, on the other hand, introduces formal verification approaches to rigorously analyze system behaviours, ensuring a higher degree of confidence in safety-critical systems.

DO-178C emphasizes that artifacts produced must meet objectives related to consistency, accuracy, traceability, and verifiability. This ensures that the outputs align with inputs and that every artifact is actionable and measurable (refer to Tables in annex A in [3]). By addressing these stringent criteria, DO-178C provides a robust foundation for developing reliable and certifiable software for airborne systems.

Figure 2.2 illustrates the DO-178C software development lifecycle, covering the progression from HLR to software coding, with inputs from the system development lifecycle (notably, the SASR). The SASR, extracted from [13], provides the initial inputs for the HLR specification, which is further refined to create the software architecture and, subsequently, the LLR. In the figure, each development stage includes an example requirement to demonstrate traceability

through the lifecycle from the SRATS-1 to the LLR-1. This approach emphasizes the critical flow from system-level considerations to the detailed software implementation, ensuring that all aspects of the system are captured.

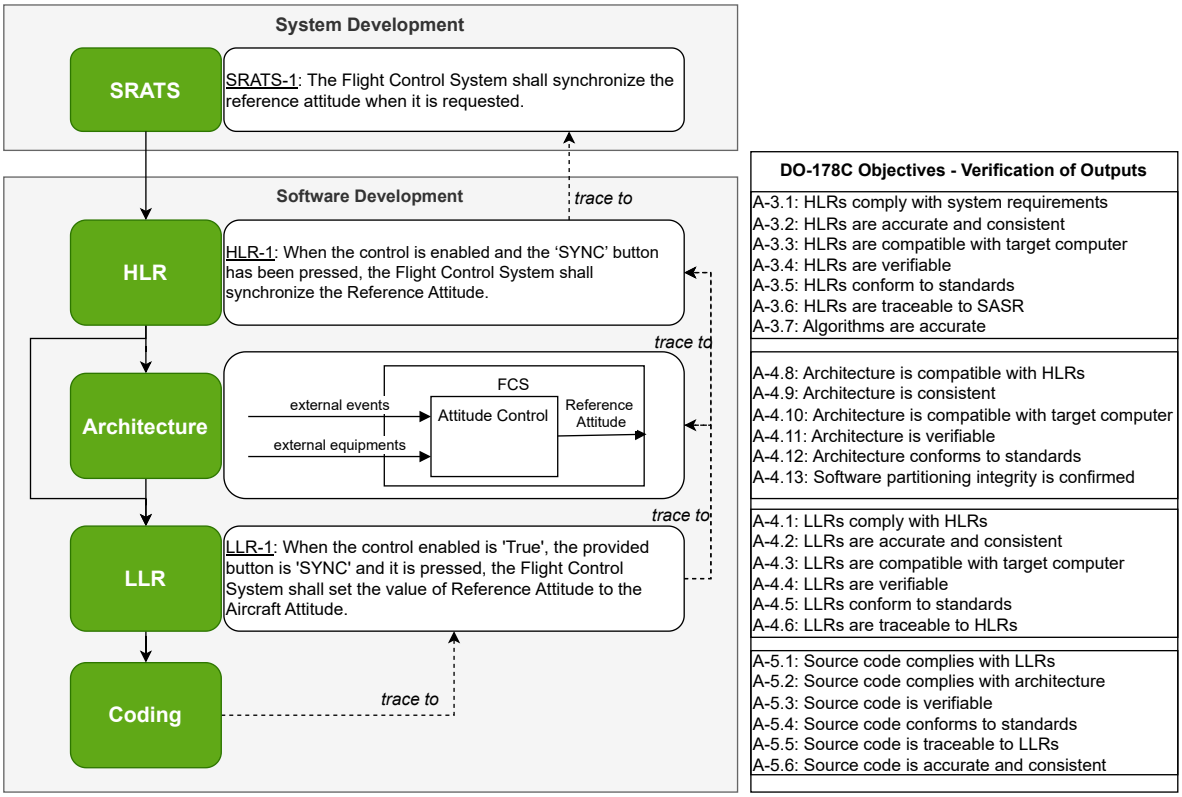

Figure 2.2 Overview and an example of DO-178C processes and their verification outputs objectives

Alongside each output artifact (HLR, architecture, and LLR), the associated DO-178C verification objectives that must be fulfilled are also shown. These objectives range from verifying that HLRs are compliant with system requirements (Objective A-3.1) to ensuring that the final source code conforms to standards and is accurately traceable to LLRs (Objectives A-5.4 and A-5.5). By explicitly linking each development phase with the corresponding verification objectives, the image highlights the rigorous and structured approach required by DO-178C to achieve software certification. This comprehensive view of the development and verification processes underscores the importance of maintaining consistency and traceability throughout the software lifecycle, a necessity for certifying safety-critical systems like those in aviation.

2.1.2 ARINC653

ARINC 653, officially titled "Avionics Application Standard Software Interface" [14], is a key standard developed by ARINC to support the development and certification of avionics systems. It establishes a standardized interface for real-time operating systems (RTOS) used in avionics, specifically within Integrated Modular Avionics (IMA) environments. By providing a well-defined framework, ARINC 653 simplifies the design and integration of avionics software components, enabling system designers to achieve compliance with stringent safety and reliability requirements mandated by aviation authorities. The standard plays a key role in fostering software component modularity, reusability, and interoperability—all of which are essential for cutting expenses and expediting the development of modern avionics systems.

Fundamentally, ARINC 653 relies on software partitioning for safety-critical software, which is defined by isolating software components into independent, distinct partitions. This partitioning improves the resilience and safety of the system by ensuring that errors or failures in one partition do not impact others. Furthermore, ARINC 653 specifies mechanisms for time and space partitioning, allowing each software partition to operate within a predefined time slice and memory space. For avionics operations to be executed predictably and meet real-time performance requirements, deterministic scheduling is essential. To facilitate safe and effective data transfer across partitions while preserving their isolation, the standard also specifies Inter-Partition Communication (IPC) protocols. These characteristics combine to make ARINC 653 a crucial framework for facilitating the creation of intricate, multipurpose avionics systems.

The adoption of ARINC 653 offers substantial benefits for avionics software development, especially in systems demanding high safety and certification levels. Its standardized interface allows developers to create modular software components that can be reused across various systems or platforms, minimizing the need for redundant development efforts. This modularity, along with stringent partitioning and scheduling mechanisms, streamlines the integration of software components from different suppliers, promoting greater collaboration within the aerospace industry. Additionally, ARINC 653's structured framework facilitates compliance with regulatory standards such as DO-178C, simplifying the certification process and ensuring that safety-critical systems meet stringent airworthiness requirements. By addressing key challenges in avionics development, ARINC 653 has become a fundamental element in modern aerospace engineering, enabling the creation of scalable, efficient, and reliable avionics systems.

2.2 Formal Methods

Formal methods encompass mathematical techniques and tools aimed at specifying, modelling, and verifying systems rigorously [15]. By using mathematically specified notations and structures, these methods ensure a clear and unambiguous depiction of a system's behaviour, bringing precision to system design. In this way, formal approaches improve comprehension, eliminate uncertainty, and facilitate the construction of robust and dependable systems throughout their development lifecycle. Their use is particularly beneficial in sectors where reliability and safety are critical.

Formal techniques are fundamentally based on the development of formal models, abstract representations of system elements, and formal analysis, a mathematical technique for confirming that these models satisfy specified properties [16]. Formal models, described using notations with mathematically exact syntax and semantics, make detailed and precise representations of systems possible. These models are then used in formal analysis to guarantee that crucial characteristics, including functionality and safety, are continuously met. This process provides a foundation for proving the correctness of a system while minimizing reliance on human intuition.

One of the main advantages of formal approaches is their adaptability. They can be used at several phases of the system development lifecycle, such as requirement specification, high-level design verification, low-level implementation refinement, and even operational system analysis [17]. A crucial component of these techniques is formal verification, which entails creating proofs to show that a system meets its specifications. Formal verification ensures that a system's actual behaviour matches its intended behaviour by analytically examining every possible case.

Despite their benefits, the application of formal methods is not without challenges. Models created through these methods often focus on specific properties of a system, which means they may not comprehensively address every aspect of the design. Therefore, it is essential to understand the scope and limitations of a formal model. Properties outside its scope must often be addressed through complementary techniques or traditional approaches like those outlined in standards such as DO-178C. Nonetheless, when used effectively, formal methods can significantly enhance software development by providing systematic and repeatable analysis [7].

In safety-critical domains, formal verification methods such as model checking and theorem proving are particularly prominent [17]. Model checking [12], in which its process is depicted in Figure 2.3, systematically explores all possible states of a system formally modelled to

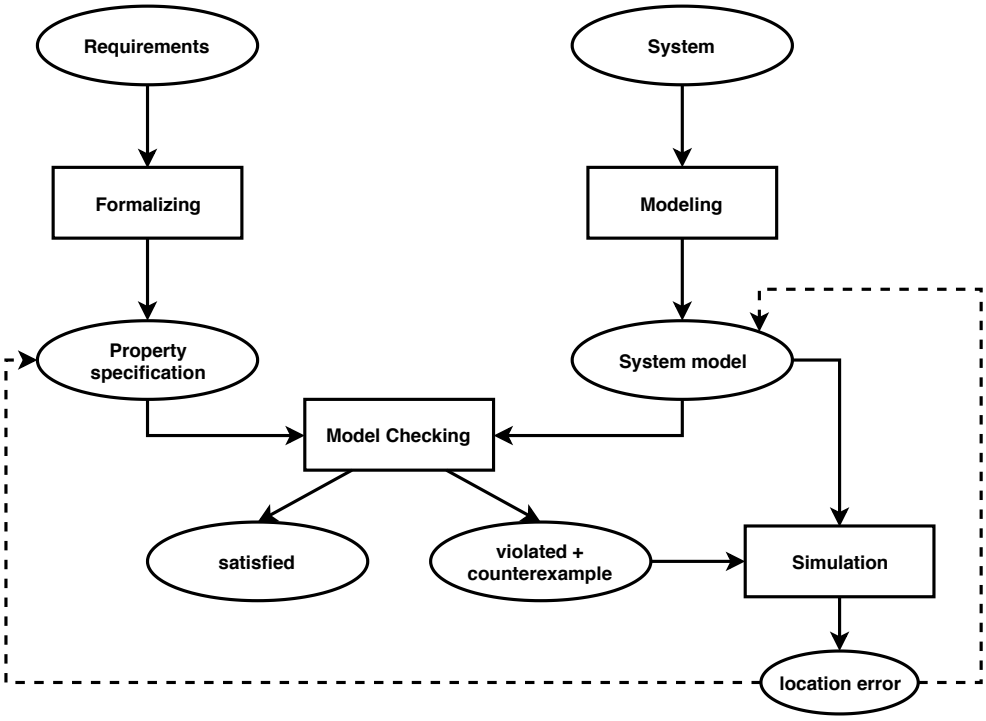

Figure 2.3 Schematic view of model checking technique (*Source*: [12]).

verify properties, while theorem proving uses logical reasoning to prove a system's correctness. Recently, runtime verification has gained attention for its ability to monitor system behaviour in real-time and ensure compliance with safety requirements. These methods have proven to be of great value in industries such as aerospace and automotive, where system failure can have catastrophic consequences.

To summarize, formal methods provide a mathematically based strategy for guaranteeing system accuracy and dependability. Formal models, precise notations, and rigorous analysis techniques are integrated to facilitate the design and verification of systems that satisfy strict functional and safety requirements. While their application requires careful consideration of scope and complexity, the benefits they bring to safety-critical systems make them an indispensable part of modern system engineering practices.

2.3 Model-Based Engineering

Model-Based Engineering (MBE) is a systems and software engineering paradigm that emphasizes the use of models as the central artifacts throughout the development lifecycle [18, 19]. In contrast to conventional development techniques, such as textual requirements specification, MBE uses formalized models to depict a system's requirements, behaviour,

and structure. By doing early analysis, validation, and verification, these models help engineers eliminate uncertainty and enhance stakeholder communication. MBE is an effective method for managing large-scale engineering projects because it offers a high-level abstraction of complex systems, which makes design decisions, requirements traceability, and iterative development easier.

MBE has major benefits in safety-critical domains like avionics, where strict safety and reliability standards must be fulfilled. Formal models guarantee compliance with regulatory standards like DO-178C and enable the early detection of design defects. For instance, MBE facilitates tasks that are essential to guarantee system compliance and robustness, such as automated code generation, model-based requirements specifications, and formal verification. Safety-critical systems can be simulated and analyzed under a variety of conditions using tools and approaches that are in line with MBE, such as Model-Based Development (MBD) and Model-Based Testing (MBT), which lowers the possibility of expensive rework. In order to prove compliance with certification procedures, MBE also encourages traceability between high-level requirements and lower-level implementations.

Domain-Specific Modeling Languages and model transformations are two fundamental concepts in MBE that are crucial to this thesis. These concepts are tailored to address specific domain requirements, enhancing clarity and reducing ambiguity. They play a critical role in effective communication among stakeholders, especially in the development of safety-critical systems. In such domains, these technologies streamline development by automating complex tasks, ensuring consistency, traceability, and compliance with rigorous standards, thereby minimizing errors and improving the overall quality of the system.

2.3.1 Domain-Specific modelling Language

A Domain-Specific Modeling Language (DSML) is designed to address the unique needs of a particular domain by offering specialized constructs, concepts, and notations tailored to that domain [20]. Unlike general-purpose languages like the Unified Modelling Language (UML) [21], which are intended to model systems across a wide range of domains, DSML focuses on solving specific domain problems. By incorporating domain-specific vocabulary and semantics, DSML enables more precise communication among stakeholders, reduces ambiguity, and simplifies the complexity inherent in system development [22]. Additionally, DSML improves production efficiency by automating domain-relevant tasks and facilitating validation and analysis.

The development of a DSML requires expertise in the target domain and involves three critical components, as referred to in Figure 2.4: abstract syntax, concrete syntax, and

semantics [23]. These components collectively define the structure, appearance, and meaning of the language, ensuring it meets the needs of its intended users. Together, they provide a robust framework for expressing domain-specific concepts while maintaining the precision and consistency required for complex system development.

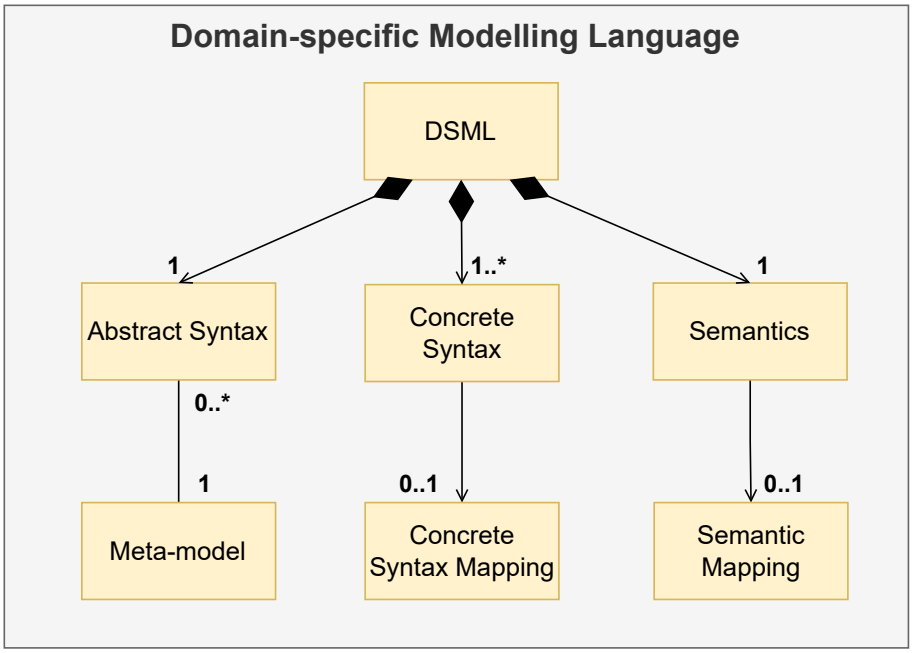

Figure 2.4 Overview of the DSML (Source: [24])

The abstract syntax forms the backbone of a DSML, defining its structural rules and elements. This is typically represented using a metamodel, which serves as a high-level description of the domain concepts, their relationships, and the constraints governing them. UML class diagrams are often used to define metamodels, offering a formal representation of the domain's structure [23]. The metamodel ensures that all models created with the DSML adhere to consistent rules and accurately represent the domain.

While the abstract syntax provides the structural foundation, the concrete syntax focuses on how the language is represented to users. This can take the form of textual notations, graphical symbols, or a combination of both. The concrete syntax bridges the gap between the formal structure defined by the abstract syntax and the usability of the language for practitioners [25]. By providing intuitive and accessible representations, the concrete syntax enhances the user experience and facilitates the practical application of the DSML.

Another crucial aspect of DSML is its semantics, which defines the meaning of the language

elements. Semantics can be expressed in natural language or formalized using mathematical logic. This layer enables reasoning about the models, such as verifying constraints, analyzing relationships, or simulating system behaviour [23]. In safety-critical domains, where precision and correctness are paramount, well-defined semantics are particularly valuable for ensuring the validity of models and supporting certification processes.

DSMLs are widely recognized as powerful tools for streamlining system development, particularly in specialized and safety-critical domains such as avionics. By providing tailored abstractions, they allow engineers to focus on domain-specific challenges while reducing cognitive overhead. Furthermore, DSMLs support domain-specific assessments, simulations, and automated transformations, enhancing productivity and fostering innovation. As a result, they play a critical role in modern MBE methodologies, including those applied in this thesis.

2.3.2 Model Transformation

Model transformation is a cornerstone of MBE, providing mechanisms to convert models from one representation to another systematically. This process plays an integral role in refining and specializing software artifacts across various phases of the software development lifecycle, including requirements engineering, design, and code generation. By automating the transition between different levels of abstraction or representations, model transformation enhances productivity, ensures consistency, and fosters traceability throughout the development process [26].

According to [27], model transformation can be defined as "*the automatic generation of a target model from a source model, according to a transformation definition.*" A transformation definition comprises a set of rules that specify how constructs in the source model can be mapped to constructs in the target model. These rules provide the foundation for systematically translating high-level specifications into lower-level artifacts, ensuring alignment between the different representations of the system under development.

Model transformations offer several advantages [28]. First, they bridge the gap between abstract requirements and concrete implementations by systematically refining higher-level models into executable artifacts. Second, they improve traceability and consistency between models at different abstraction levels, which is critical for safety-critical systems. Third, transformations reduce manual effort by automating repetitive tasks, such as code or documentation generation, thereby minimizing human error and accelerating the development process. These benefits make model transformation an indispensable part of modern software engineering practices.

Transformations are typically classified into two categories: model-to-model (M2M) transformations and model-to-text (M2T) transformations. M2M transformations focus on converting models between different formats or abstraction levels, such as transforming a UML model into an AADL representation. In contrast, M2T transformations generate textual artifacts like source code or configuration files directly from models. Each type of transformation serves distinct purposes within MBE, often working in tandem to ensure the seamless progression from high-level design to final implementation.

In safety-critical domains like avionics, model transformations are particularly valuable. They enable the systematic integration of standards like DO-178C, ensuring compliance with regulatory objectives while maintaining consistency across artifacts. Moreover, by employing transformation definitions that encapsulate domain knowledge, developers can enforce correctness and completeness in the transformation process. As a result, model transformations not only streamline development but also enhance the reliability and safety of the resulting software systems. This thesis leverages model transformation to bridge the gap between domain-specific modelling languages and architecture descriptions, ensuring consistency and traceability throughout the software development lifecycle.

2.4 Tools and Languages

2.4.1 AADL

The Architecture Analysis & Design Language (AADL) [29] is a standard modelling language for designing and analyzing complex, real-time, embedded systems. It provides a robust foundation for representing the architecture of such systems, encompassing both software and hardware components, their interactions, and associated properties. AADL is a well-known tool in industries including avionics, aerospace, and automotive that provides accurate modelling and analysis capabilities, making it essential for guaranteeing system safety and dependability.

AADL is designed as a component-based modelling language, a feature that offers several practical advantages. Using a combination of textual and graphical representations enables engineers to specify both functional and non-functional features of a system, making it flexible enough to accommodate a range of use cases. The language's notations allow detailed descriptions of processes, communication mechanisms, and the runtime architecture of safety-critical, real-time, and embedded systems. By providing well-defined semantics, AADL assures that architectural models can be analyzed, allowing engineers to assess system performance, resource allocation, and reliability effectively [30].

The language is structured around key concepts such as components, connections, and properties. A relationship between the components and their attributes is depicted in Figure 2.5, either in the form of a hierarchy, references, or more information about the properties that the component may possess. The core of AADL models are components, which stand in for distinct parts of the system architecture, such as hardware, software modules, and processors. These elements are organized into types and implementations, where the implementation outlines the internal structure of the component, and the type declaration defines its interface and category. A precise depiction of system interactions is ensured by connections between components, which capture information flow, communication, and relationships. Conversely, properties can characterize non-functional aspects, including resource usage, timing restrictions, and fault-tolerance specifications [29].

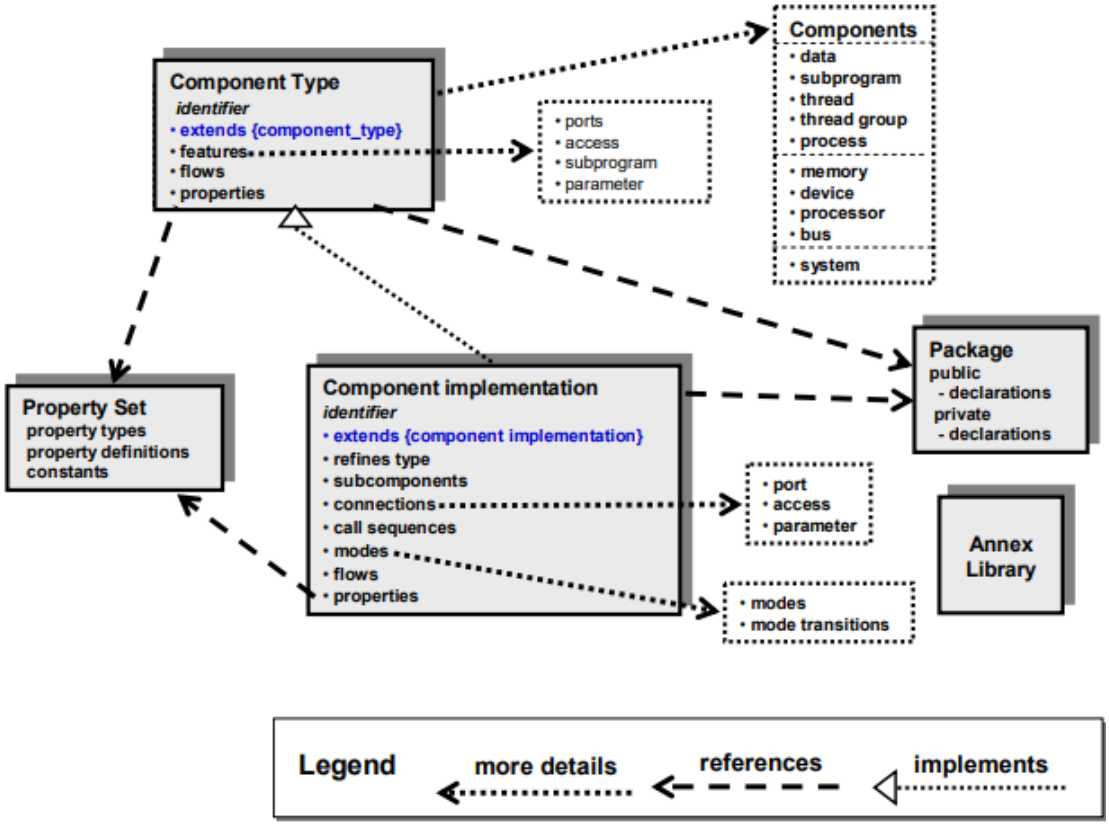

Figure 2.5 AADL summary elements. Source: [29]

One of AADL's special advantages is its extensibility through annexes that supplement the fundamental modelling principles, such as the Behavior Annex and the Error Model Annex. These annexes make comprehensive explanations of fault-tolerant settings, error handling, operational modes, and system behaviour possible. For embedded systems, which frequently have to strike a balance between handling event response and continuous control processing,

the capacity to model runtime characteristics and operational modes is very beneficial.

AADL's precise execution semantics and support for modelling large-scale systems make it indispensable for diverse analysis needs. As evidence of its effectiveness, the language makes it easier to spot any design problems early on, reducing the need for later, expensive rework. Additionally, AADL models can be analyzed for critical properties, such as performance, resource contention, and safety compliance, making it a powerful tool for industries where certification and reliability are paramount [31].

In summary, AADL bridges the gap between system design and analysis by providing a formal and comprehensive approach to architecture modelling. Its integration of textual and graphical elements, along with extensibility through annexes and support for automated tools, empowers engineers to create, analyze, and refine system architectures with confidence. This makes AADL an invaluable asset in the development of safety-critical systems, ensuring that they meet the rigorous demands of certification and operational safety.

2.4.2 DO-178C software development tools

Tools play a crucial role in streamlining the development process, ensuring compliance, and supporting certification efforts in DO-178C software development. These tools can be categorized based on their primary functions: requirements management, code analysis, and certification assistance.

Requirements Management Tools:

Effectively managing requirements is essential in DO-178C projects to ensure traceability and compliance. Tools like IBM DOORS [32], LDRA TBManager [33] and Jama Connect[1], for example, are widely used to capture, organize, and link system and software requirements. These tools provide traceability matrices that help developers verify that all requirements are accounted for throughout the lifecycle and that any changes are effectively managed. Additionally, they facilitate collaboration between stakeholders and generate comprehensive documentation to support certification audits.

Code Analysis Tools:

Code analysis tools are crucial for verifying the correctness, consistency, and compliance of the source code with the LLRs and coding standards. Tools like LDRA Testbed[2], Vector-

[1] https://www.jamasoftware.com/
[2] https://ldra.com

CAST[3], and Polyspace[4] are frequently used to perform static and dynamic analysis, ensuring adherence to industry standards like MISRA C and the requirements defined by DO-178C.

Certification Assistance Tools:

Certification assistance tools support the generation of evidence required for DO-178C certification audits. Tools such as the LDRA tool suite and Rapita Verification Suite[5] help automate the verification and validation processes, generate traceability matrices, and prepare comprehensive documentation for certification purposes. By using these qualified tools, development teams can produce certified evidence that meets DO-178C objectives systematically, reducing manual effort and the potential for errors. These tools are particularly valuable in preparing for audits and demonstrating compliance with the certification authority.

Given the diverse tool requirements for DO-178C software development, covering requirements management, code analysis, and certification assistance, the LDRA tool suite was chosen for this thesis as it effectively fulfills all three primary functions within a single, integrated environment. It is a set of tools including four core components: LDRA Testbed, TBvision, TBrun, and TBmanager [34].

TBmanager [33] is the primary coordination tool in the LDRA ecosystem, guaranteeing smooth traceability throughout all development stages. It provides an integrated environment for managing compliance workflows by connecting requirements, design artifacts, source code, tests, and verification outcomes. By preserving bidirectional traceability, as depicted in Figure 2.6, TBmanager lessens the work needed to create certification artifacts and assists organizations in proving alignment with regulatory goals. Furthermore, its integration with external development and lifecycle management tools improves its versatility across various software engineering processes.

[3]https://www.vector.com/
[4]https://www.mathworks.com/products/polyspace.html
[5]https://www.rapitasystems.com/products/rvs

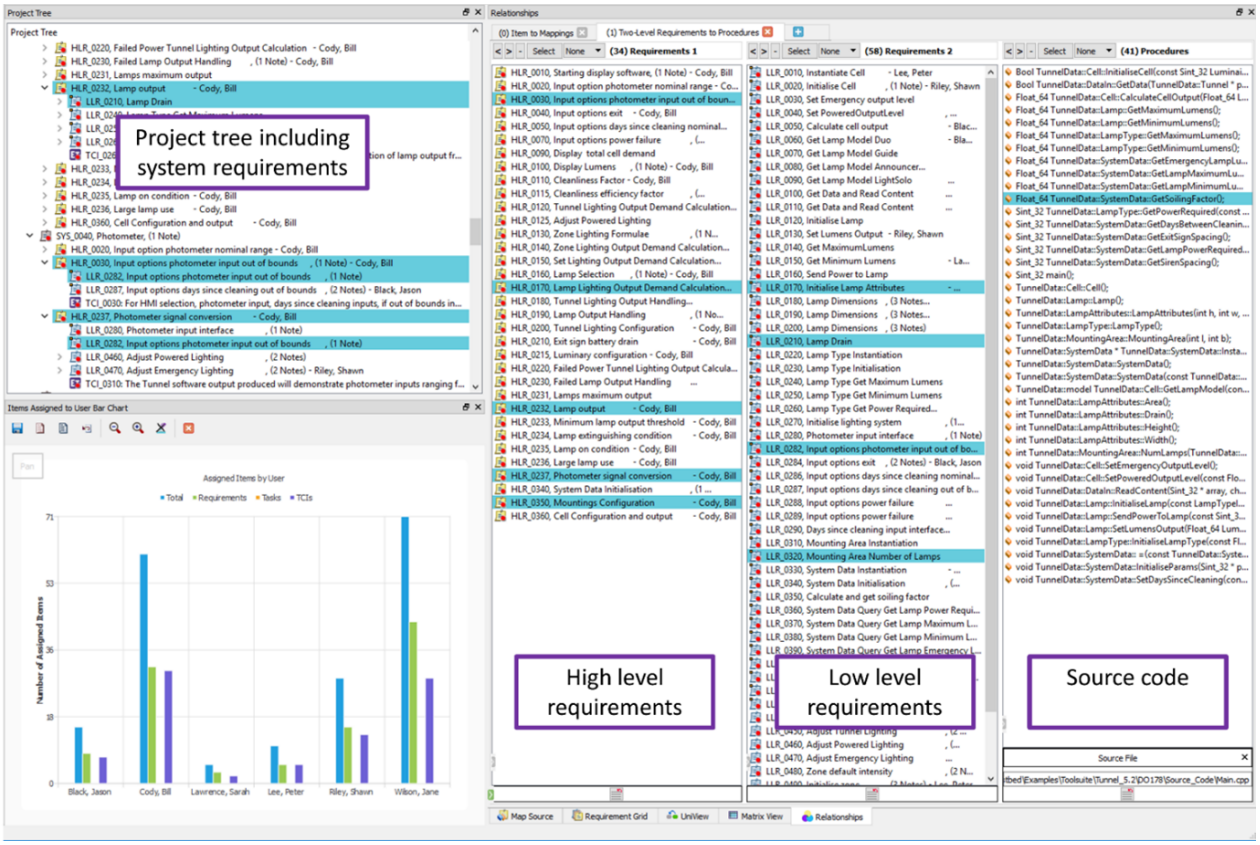

Figure 2.6 TBmanager view showing the traceability between different requirement levels and source code. Extracted from `https://ldra.com/products/tbmanager`

A fundamental basis for both static and dynamic analysis is the LDRA Testbed. It conducts deep code inspections using its proprietary parsing engine to find possible flaws, security vulnerabilities, and coding standard violations. By providing a visual depiction of software quality metrics and compliance status, TBvision [35] supplements this capability and improves the analysis process. Developers can effectively evaluate software integrity, identify severe defects early, and uphold compliance with strict regulatory requirements by combining these tools. As an example, Figure 2.7 shows the TBvision window in which the result of the code review points out the violations found (in the right-hand window).

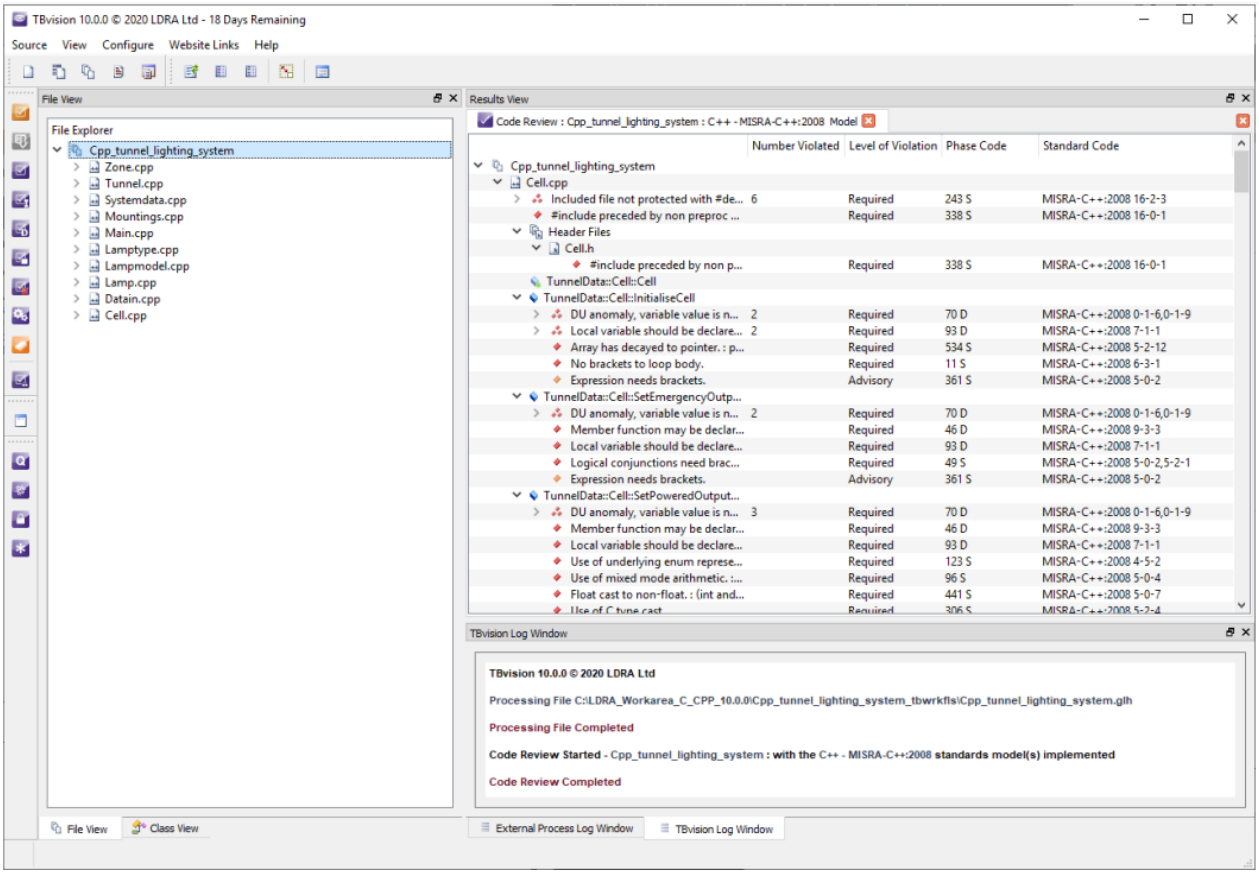

Figure 2.7 TBvision view showing code review violations. Extracted from [35]

TBrun [36] plays a critical role in software testing by automating unit and integration tests. It makes it easier to create and run test harnesses, enabling developers to assess how software behaves in different scenarios. To ensure thorough verification, TBrun facilitates testing in various settings, including host-based simulations and real hardware targets. Its instrumentation methods also offer insights into structural coverage, including Modified Condition/Decision Coverage (MC/DC) metrics, which are crucial for certification in high-integrity fields like avionics.

A structured framework for creating safety-critical software is offered by the LDRA tool suite, which unifies lifecycle traceability, test automation, and static and dynamic analysis. Its all-encompassing strategy lowers development costs and time-to-market while simultaneously streamlining the certification process and enhancing software quality and security.

2.4.3 Formal Methods Tooling

This thesis employs a suite of formal methods tools integrated across different phases of the software development lifecycle to ensure rigour and correctness in safety-critical avionics systems. The tools were selected to align with the abstraction levels of the development process:

- **Alloy Analyzer** [37] is a lightweight formal modelling tool that uses relational logic to specify and analyze system properties. It allows users to model structures and behaviours with precision, and automatically checks the consistency of specifications by generating counterexamples within a bounded scope.

  In this thesis, Alloy was used to formally specify and verify the high-level and low-level requirements of the avionics system. Requirements were expressed in the Alloy language, and the Analyzer was employed to check for consistency, completeness, and the absence of unintended behaviours. This enabled early validation of system logic before design and implementation.

  Alloy was chosen for its expressiveness in modelling complex relationships, its ease of use compared to heavier theorem provers, and its automated analysis capabilities. It was particularly suited for verifying the logical soundness of requirements models without requiring extensive proof engineering.

  An example of how to use Alloy in the software development cycle is provided below. By considering the example given in the sub-section 2.1.1, the HLR "*When the control is enabled and the 'SYNC' button has been pressed, the Flight Control System shall synchronize the Reference Attitude*" is specified in Alloy as follows:

```
sig FlightControlSystem {
controlEnabled: one Bool,
syncButtonPressed: one Bool,
referenceAttitudeSynchronized: one Bool
}

pred synchronizeReferenceAttitude[sys: FlightControlSystem] {
    sys.controlEnabled = True and
    sys.syncButtonPressed = True implies
    sys.referenceAttitudeSynchronized = True
}

fact HLR_Synchronization {
    all sys: FlightControlSystem |
        synchronizeReferenceAttitude[sys]
}
```

Listing 2.1 Example of Alloy Specification

In which:

- *FlightControlSystem* is a signature representing the system.
- Three Boolean attributes model the system's state:
  - *controlEnabled* (whether the control is active)
  - *syncButtonPressed* (whether the SYNC button is pressed)
  - *referenceAttitudeSynchronized* (whether the attitude has been synchronized)
- The predicate *synchronizeReferenceAttitude* formalizes the logic: If the control is enabled and SYNC is pressed, then the reference attitude must be synchronized.
- The *fact* block enforces that this predicate must hold for all system states.

By modelling the HLR in Alloy, the requirement is transformed from informal natural language into a precise, mathematical specification. This eliminates ambiguity by explicitly defining the system's expected behaviour using relational logic. The Alloy Analyzer automatically checks for logical consistency across all defined facts and predicates. For instance, it can identify whether contradictory constraints exist or whether the requirement permits unintended behaviours. Additionally, the analyzer can generate counterexamples when the requirement is not satisfied, helping engineers uncover edge cases or underspecified conditions. In this way, Alloy strengthens the rigour, precision, and internal consistency of the requirements, ensuring that they are both verifiable and unambiguous before proceeding to design and implementation.

- **Spin Model Checker** [38] is a widely used model checker for verifying the correctness of distributed software systems. It uses the Promela language to model system behaviour and verifies properties such as safety, liveness, and reachability through exhaustive exploration of system states.

  Spin was applied to formally verify the software architecture, ensuring that interactions between components conformed to expected communication protocols and that system-level properties such as deadlock-freedom and proper synchronization were upheld.

  It was selected for its ability to model communication between different sub-systems and perform exhaustive checks efficiently. Its proven applicability to software architecture verification and strong tool support made it well-suited for ensuring correctness at the architectural level in the avionics context.

  By also using the example provided in 2.1.1, the Spin can be applied to formally model and analyze the behaviour of the Attitude Control System (ACS), a critical subsystem responsible for providing the reference attitude. This architectural model captures

both external events (such as user interactions like SYNC button presses and control enable signals) and equipment inputs (such as aircraft sensor data indicating the current attitude).

The system operates by reacting to these inputs to update the reference attitude, which is then propagated to other flight control functions. The architecture is modelled in Promela, Spin's process modelling language, where each architectural component (e.g., UserInputHandler, SensorInterface, SynchronizationLogic) is represented as a process communicating over channels. The Spin model is depicted in 2.2

```
bool control_enabled = false;
bool sync_pressed = false;
bool reference_attitude_updated = false;

int aircraft_attitude = 0;
int reference_attitude = 0;

chan user_input = [0] of {bool, bool}; // control_enabled,
    sync_pressed
chan sensor_input = [0] of {int};      // aircraft_attitude

proctype UserInputHandler() {
    user_input! (true, true);
}

proctype SensorInterface() {
    sensor_input! 45;
}

proctype SynchronizationLogic() {
    bool ctl;
    bool sync;
    int att;

    user_input? (ctl, sync);
    control_enabled = ctl;
    sync_pressed = sync;

    sensor_input? att;
    aircraft_attitude = att;
```

```
30
31          if
32          :: (control_enabled && sync_pressed) ->
33              reference_attitude = aircraft_attitude;
34              reference_attitude_updated = true;
35          :: else ->
36              reference_attitude_updated = false;
37          fi;
38      }
39
40      init {
41          run UserInputHandler();
42          run SensorInterface();
43          run SynchronizationLogic();
44      }
```

Listing 2.2 Promela model of the Attitude Control System

With the model defined, properties are specified based on the high-level requirements (phase preceding the design phase) to verify that the architecture conforms to them. Examples of properties are:

- **Synchronization Responsiveness (Liveness Property)** "*If the control is enabled and the SYNC button is pressed, the reference attitude will eventually be updated.*"

  $\Box(control_enabled \ \&\& \ sync_pressed \rightarrow \diamond \ reference_attitude_updated)$

- **No Synchronization Without Permission (Safety Property)** "*The system shall not update the reference attitude unless the control is enabled.*"

  $\Box(!control_enabled \ \rightarrow !reference_attitude_updated)$

- **ESBMC (Efficient SMT-Based Bounded Model Checker)** [39] is a bounded model checker for C and C++ programs that leverages SMT (Satisfiability Modulo Theories) [40] solvers to analyze program behaviour. It systematically explores feasible paths within a given bound and can detect runtime errors, assertion violations, and logic flaws.

  ESBMC was employed to verify the source code generated from Simulink models. Sensor input values were replaced with symbolic representations and assumptions, allowing

ESBMC to explore execution paths and check for correctness against functional requirements, especially in safety-critical logic.

It was chosen for its compatibility with C code and support for verifying embedded software properties. Its suitability for bounded analysis made it practical for detecting defects in auto-generated code within constrained analysis timeframes.

2.5 Related Works

The development of software for avionics systems, particularly under the stringent requirements of the DO-178C standard, has been a focal point of research due to the increasing complexity and safety-critical nature of airborne systems. Recent advancements have sought to enhance methodologies by integrating formal methods, model-based approaches, and automation tools to streamline the certification process while ensuring compliance with DO-178C objectives.

2.5.1 Methodologies for Software Development and Certification in Avionics

The DO-178C standard does not mandate a specific development methodology but instead outlines objectives that must be met to demonstrate compliance with airworthiness requirements. This flexibility allows organizations to choose methodologies that best suit their projects while ensuring adherence to safety-critical standards. For many years, structured methodologies such as the Waterfall and V-Model have been widely used in avionics software development due to their sequential and well-defined processes [7].

The Waterfall methodology follows a linear progression through stages such as requirements gathering, design, implementation, testing, and maintenance [41]. Its strengths lie in its simplicity and clarity, which facilitate project planning and stakeholder communication. However, its rigid structure makes it difficult to accommodate changes once development has begun, leading to costly rework when unexpected issues arise later in the lifecycle. In contrast, the V-Model [42] enhances the Waterfall approach by introducing validation and verification steps at each phase, creating a direct correlation between development and testing. This model ensures early defect detection and improved traceability, which are critical in safety-critical domains. Nevertheless, both methodologies share a common limitation: their limited adaptability to dynamic requirements or iterative refinements, which are increasingly relevant in modern, complex systems [43].

Despite the longstanding association of the Waterfall model with DO-178C compliance, the standard does not prescribe this methodology, nor does it express a preference for any specific

lifecycle model. Projects that incorrectly claim to follow the Waterfall methodology without adhering to its principles face significant challenges, including misaligned processes and potential certification risks [7]. It is essential for projects to identify and follow the lifecycle model they actually use, ensuring that it satisfies DO-178C objectives and is documented appropriately.

Over time, new approaches have been developed to address the limitations of traditional methodologies and streamline the certification process. For example, the authors in [43] discuss integrating Agile principles into safety-critical software development. While Agile emphasizes flexibility and iterative progress, its direct application in regulated environments requires careful adaptation. It highlights how Agile methodologies, combined with structured frameworks like Scrum, offer benefits such as faster feedback loops and enhanced collaboration. However, these methods must be tailored to meet DO-178C objectives, particularly in documentation and traceability, to ensure compliance.

Another advancement in software development methodologies for avionics is the adoption of MBD techniques, as discussed in [44]. The authors propose formal verification methods and multi-physics modelling to accelerate certification processes while addressing the growing complexity of CPS. This integrated framework underscores the need for compositional "design-for-certification" methods, bridging the gap between traditional practices and modern engineering demands.

Further contributions to model-based development include [45]. It presents a DSML developed as a UML profile designed to enforce certification documentation requirements based on the DAL. This approach provides an infrastructure that ensures compliance with DO-178C and its supplements, facilitating structured documentation throughout the development lifecycle.

A case study on developing landing gear control software offers practical insights into combining DO-178C, DO-331, and DO-332 methodologies. The study in [46] emphasizes creating detailed requirements, specifications, and designs that reflect the complexity and constraints of real-world avionics systems. By making these specifications open and accessible, the authors provide a valuable benchmark for evaluating model-based approaches while highlighting the challenges encountered during the development and certification processes.

For emerging domains like eVTOL aircraft, research has explored innovative methodologies tailored to the unique challenges of flight control software. [47] demonstrates the use of incremental nonlinear dynamic inversion techniques combined with model-based development to achieve DO-178C compliance. This pragmatic approach offers a comprehensive toolchain for developing and verifying flight control software, showcasing the potential of modern tools

and techniques for certifying next-generation aviation systems.

Also, recent advancements include exploring adaptive systems and cognitive architectures for UAVs, which present unique certification challenges under DO-178C. In [48], the authors investigate methods to assess and validate the dynamic behaviour of adaptive learning agents, proposing strategies to align these cutting-edge technologies with existing certification frameworks.

These evolving methodologies reflect a rising trend of combining established, structured practices with contemporary, adaptable approaches. This integration ensures that safety-critical software development meets the stringent objectives of DO-178C while also tackling the complexities inherent in modern aerospace systems.

Integrating Formal Methods to the Software Development

The use of formal methods in avionics software development has gained attention due to their potential to enhance the verification process of safety-critical systems. Formal methods rely on mathematically defined syntax and semantics to establish properties and ensure correctness, making them particularly suitable for the rigorous demands of the DO-178C certification process. While traditional techniques like testing and reviews remain the most commonly used verification methods, they have notable limitations: they cannot exhaustively demonstrate the absence of defects, are primarily manual processes, and are most effective only in the later stages of development [17]. In contrast, formal methods offer an analytical approach, often applied during the development phase, to establish properties and detect errors early. DO-333 [16], the Formal Methods supplement to DO-178C, recognizes this potential, listing examples such as graphical, textual, and abstract models as formal tools to achieve certification objectives.

One significant work in this domain is [49], which explores the challenges and benefits of integrating formal methods into certification processes. This paper emphasizes the essential characteristics of formal methods and proposes a Formal Methods Technical Supplement to DO-178C. By outlining objectives and practical guidance for their adoption, this study aims to make formal methods a recognized and practical approach to safety-critical software certification.

The integration of formal verification and testing is further explored in [50]. This paper addresses the complementary roles of formal verification and testing in the avionics lifecycle, presenting a unified methodology that bridges the gap between these activities. Formal verification is used to validate safety-critical properties at the design level, while testing, such as

Modified Condition/Decision Coverage (MC/DC), ensures these properties are maintained during implementation. The results from these processes can be used as certification evidence, demonstrating the value of cross-fertilization between formal methods and traditional verification techniques.

Despite their strengths, formal methods tools face practical challenges, as highlighted in [51]. One key concern is tool soundness — ensuring that verified properties are truly correct. This study examines two major classes of formal tools, model checkers and static analyzers, and identifies threats to their soundness, such as tool fallacies and failure modes. Addressing these issues is essential to prevent misplaced confidence in verified systems and to maintain the rigorous standards required by DO-178C.

The practical application of formal methods in avionics certification is exemplified in [52, 53], which presents three case studies involving a dual-channel Flight Guidance System. These studies showcase the use of theorem proving, model checking, and abstract interpretation, demonstrating how each technique can satisfy different certification objectives. By illustrating the strengths and weaknesses of these approaches, the paper provides a realistic perspective on their applicability to avionics systems.

A related study, [54] delves into the challenges of qualifying formal methods tools under the DO-330 framework. The study proposes mitigations for obstacles in tool qualification, explores the feasibility of generating proof certificates to validate tool claims, and examines the potential to qualify proof certificate checkers instead of the tools themselves. These efforts underscore the importance of ensuring trust in the tools used for certification.

In the context of autonomous systems, [55] evaluates the feasibility of applying formal methods to certify Unmanned Aircraft Systems (UAS). The authors use PROMELA and the SPIN model checker to verify a basic UAS control system against a subset of the Rules of the Air. They then extend their approach using the Gwendolen agent language and the AJPF model checker to validate advanced autonomous behaviours. This work demonstrates the adaptability of formal methods to emerging domains like autonomous UAS, paving the way for their broader application.

Together, these contributions highlight the growing role of formal methods in the avionics software lifecycle. By addressing challenges, proposing integrated methodologies, and demonstrating practical applications, formal methods continue to advance the state of the art in safety-critical system certification under DO-178C.

Integrating Automated Tools to the Software Development

The software development life cycle for avionics systems is inherently complex, with stringent requirements for compliance, traceability, and verification. Meeting DO-178C objectives requires rigorous processes, such as software verification and certification evidence generation. These processes, while essential, are resource-intensive and prone to human error when done manually. Automated tools have emerged as vital enablers, aiding developers in activities like testing, verification, traceability, and evidence management, thereby streamlining the path to certification. By reducing manual effort and enhancing accuracy, these tools address critical challenges in achieving compliance with the DO-178C standard.

One notable contribution in this domain is the SHERLOCK framework, as described in the paper [56]. SHERLOCK leverages advanced traceability techniques, including information retrieval and machine learning, to automate the generation and management of traceability links. Unlike traditional methods, it ensures that traceability criteria are analytically satisfied, supporting continuous and efficient certification workflows.

A complementary effort is detailed in [57], which presents a modular tool designed to enhance development efficiency while maintaining process conformance. Built using MATLAB and Simulink, this tool integrates features like modular software management, automated model and code verification, and artifact traceability. It has been successfully applied to projects ranging from flight control systems to battery management systems.

For software-defined radio systems, [58] outlines a methodology that integrates MBD with automated testing. This approach accelerates compliance with the dynamic testing requirements of DO-178C by combining model-based verification with rigorous unit and integration testing, as demonstrated in an Instrument Landing System (ILS) case study.

Addressing the certification challenges for autonomous systems, [59] presents a hybrid approach combining formal verification and flight simulation. The study emphasizes the utility of model checking for verifying rational agent-based systems, providing examples of safety properties derived from regulatory rules like the "Rules of the Air."

The integration of artificial intelligence in avionics is explored in [9]. This work proposes a structured methodology for certifying intelligent systems by combining state-of-the-art techniques in AI and DO-178C compliance. It highlights the potential of automated tools to enable UAV autonomy while meeting safety-critical requirements.

For lean and efficient development processes, [8] targets startups and resource-constrained organizations. The custom workflow it proposes focuses on automation and selective compliance, enabling higher-quality software development with reduced effort, particularly for

unmanned aircraft systems and urban air mobility platforms.

[60] introduces a software platform for generating integration test data and associated documentation. The platform streamlines testing processes with increased efficiency, providing a practical solution for software verification in critical systems.

Managing complex certification evidence is addressed in [61]. Employing the Rapid Assurance Curation Kit enables automated compliance tracking and evidence curation, facilitating the periodic review of certification progress and the generation of comprehensive certification packages.

Discussion

The growing importance of UAVs in various industries has highlighted the need for strong certification standards to ensure their safety and reliability. UAVs must use software that conforms to strict safety standards like DO-178C, which offers a structured method for building and certifying safety-critical systems, since they operate in dynamic, challenging environments. Although DO-178C places a strong emphasis on safety and dependability, there are particular difficulties when applying it to the development of UAV software. These challenges include extensive documentation and traceability demands, adapting the framework to accommodate UAV-specific features such as autonomous and sophisticated components, and addressing the complexities of verification and validation in unpredictable, real-world operational scenarios. These demands frequently result in considerable time and cost challenges, complicating compliance efforts even further.

The certification of adaptive UAV systems is one particularly difficult aspect, as explored in [47, 48]. The inherent unpredictability of these systems' behaviour in reaction to changing conditions makes it challenging to ensure their functional completeness, testability, and compliance with DO-178C criteria. Although rigorous software property specification and verification are made possible by formal methods, incorporating these approaches into current processes is still hard, particularly for companies with limited resources.

The literature identifies formal methods and automation as viable approaches to these problems ([17, 49–59]). While automation simplifies crucial procedures like requirements traceability, structural coverage analysis, and code creation, formal approaches add mathematical rigour to software correctness verification. Significant gaps still exist, though, especially in practically integrating these cutting-edge methods into industry workflows and the certification of adaptive systems. To assure the usability and efficacy of these solutions, closing these gaps will necessitate a multifaceted strategy that incorporates sophisticated methodology,

reliable tools, and real-world validation.

Several approaches have been developed to ease these difficulties and improve the certification procedure (e.g. [43–46]). MBD has gained popularity due to its ability to automate code production and increase traceability, hence decreasing manual work and increasing consistency. Automated verification technologies make testing and validation even more efficient, and early attention to safety-critical factors guarantees compliance with DO-178C objectives. Building on these advancements, our proposed approach balances accuracy and efficiency by combining automated tools, like the LDRA Tool Suite, with formal techniques. Our methodology tackles certification issues while preserving thorough evidence creation and traceability across the software lifecycle by fusing the operational advantages of automation with the rigorousness of formal verification. This integrated approach not only supports compliance with DO-178C but also facilitates the certification of advanced, adaptive UAV systems, paving the way for safer and more reliable UAV operations.

One of the key advantages of the proposed methodology, presented in Chapter 4, is the integration of qualified tools such as LDRA to systematically drive the development process and ensure compliance with DO-178C objectives. Unlike other approaches that rely on non-qualified or ad-hoc methods, the use of such tools guarantees that certified evidence is generated in an organized and traceable manner, which is essential in documentation-heavy processes like DO-178C certification. Additionally, while many methodologies in the literature apply formal methods to only specific phases of development, our approach leverages formal techniques across all phases. This comprehensive application allows for the early detection of issues, reducing costly rework and enhancing overall software reliability.

2.5.2 Approaches for Specifying Software Requirements in Safety-Critical Systems

The specification of software requirements is a critical aspect of developing safety-critical systems (SCS), as it forms the foundation for subsequent design, implementation, and verification activities. Various approaches have been adopted for capturing and managing requirements, ranging from traditional textual methods to advanced model-based techniques.

Traditional Approaches to Requirements Specification

Traditional methods for specifying software requirements often rely on natural language, typically written in text documents, spreadsheets, or tools such as IBM Rational DOORS [32]. These approaches are widely used in the industry [46, 62] due to their simplicity and ease of

communication with stakeholders. A survey conducted by [63] reveals that these methods remain the dominant practice in the field. However, the study also highlights the emergence of new research trends aimed at improving requirements specification, particularly in the context of safety-critical systems.

Despite their prevalence, natural language-based methods have inherent limitations. They are prone to ambiguity, inconsistency, and imprecision, which can lead to misinterpretations and errors in subsequent phases of the software development lifecycle [64]. To address these shortcomings, alternative approaches based on modelling languages, particularly UML, have gained traction in the industry and academia [63, 65–70]. These approaches aim to provide more structured, precise, and analyzable representations of software requirements.

Emerging Graphical and Textual Notations

One such approach is the Requirements Definition and Analysis Language (RDAL) [62], which offers both graphical and textual notations for expressing, analyzing, and validating software requirements. While RDAL supports traceability and facilitates requirements management, it does not explicitly align with the DO-178C standard, limiting its applicability in certified avionics systems.

Similarly, SpeAR [67], a tool designed to conform with DO-178C, enables the capture and analysis of requirements. However, it lacks mechanisms for enforcing critical information for HLRs, such as providing justifications for derived requirements, which are essential for compliance with DO-178C.

The Requirements State Machine Language (RSML) [65] was developed to express requirements in an intuitive format for aircraft systems. While it provides a straightforward means of representing requirements, it suffers from incomplete formalization and issues such as mixing design elements with HLRs—an approach discouraged by DO-178C. To address these issues, RSML^{-e} [66] was introduced, adding features to enhance traceability and formalization. However, its application in avionics certification remains constrained by the rigorous demands of DO-178C, such as the requirement for robust traceability between artifacts.

UML-Based Modeling Approaches

The growing interest in MBE has driven the evolution of UML-based modelling languages tailored for safety-critical systems. Three significant contributions in this domain highlight the use of DSMLs and UML profiles to address DO-178C objectives.

SpecML [68], a DSML built as a UML profile extending SysML, incorporates features based

on the Property-Based Requirement theory [71]. SpecML facilitates capturing requirements in natural language and supports analysis and testing activities aligned with DO-178C. However, it falls short in addressing detailed safety-related requirements, such as identifying system threats or accounting for software levels, which are crucial in safety-critical systems.

Another UML-based approach by [45] introduces a modelling language aligned with DO-178C, incorporating concepts like software levels and rationale. While this work addresses some key aspects of DO-178C, it still lacks mechanisms for capturing detailed safety-related requirements critical for SCS development.

SafeUML [69, 70] focuses on safety aspects of DO-178B by extending UML to capture safety requirements. Although it effectively captures some safety considerations, it does not adequately support traceability between HLRs and the SRATS objectives outlined in DO-178C. Additionally, SafeUML does not enforce a clear separation between specification and design, which is strongly recommended by the standard.

Discussion

Natural language and conventional approaches are commonly used to establish HLRs in aviation projects, owing to their accessibility and ease of communication. However, these methods frequently suffer from ambiguity and imprecision, which might result in requirements that are not fully met and make further analysis more difficult. Many languages, such as formal specification languages, which offer greater rigor and clarity, have been developed to solve these restrictions. Despite their benefits, these formal approaches need a high level of expertise from engineers, providing barriers to general use. An alternative is provided by UML-based DSMLs, which combine constraints to allow consistent analysis with graphical notations for intuitive communication.

A number of UML-based methodologies have presented DSMLs customized for DO-178, with a focus on traceability and structured communication. However, these approaches frequently lack the resources to specifically address safety concerns, which are a crucial part of DO-178C. Engineers' capacity to fully evaluate risks and hazards associated with safety during the software development process is reduced by this absence. SafeUML has many shortcomings despite its efforts to incorporate safety considerations within its standards. It does not fully align with the objectives of DO-178C, leaving engineers without clear guidance on achieving compliance. Furthermore, the DO-178C criterion of preserving a distinct division between requirements and design concepts is violated by the merging of requirements and design concepts within SafeUML.

In light of these difficulties, it is nevertheless crucial and difficult to specify HLRs for safety-critical systems, especially in ACPS. An effective specification that satisfies the exacting certification requirements of standards such as DO-178C must guarantee clarity, accuracy, and traceability. Although current methods offer insightful information, they frequently fail to fully address safety concerns or promote smooth cooperation across interdisciplinary teams.

Considering the limitations of the discussed approaches, Chapter 5 in this thesis presents a modelling language created especially for defining HLRs in ACPS software development to overcome these limitations. This modelling language, which is based on the objectives of DO-178C, emphasizes the accurate capture of safety-related requirements, enhancing stakeholder communication and analysis. Our strategy fills important gaps in current approaches by improving the traceability of requirements and aligning with certification goals, providing a strong basis for creating safety-critical systems in the aviation industry.

2.5.3 Transitioning from Requirements to Software Architecture

The development of safety-critical systems, such as those in avionics, is governed by stringent regulations due to the severe consequences of potential system failures. Every step in the software development lifecycle, from requirements definition to system deployment, must adhere to rigorous guidelines to ensure reliability and safety. A cornerstone of this process is the seamless transition from requirements to software architecture, which plays a pivotal role in shaping the system's design and functionality [7]. This transition demands not only precision and thorough documentation of requirements but also alignment of the resulting architecture with safety-critical standards and objectives. However, this activity often depends heavily on the expertise of the software architect [72], making it a challenging and subjective task.

The challenge of bridging the gap between requirements and architecture has led to the development of various methodologies, tools, and frameworks. [73] provides a comprehensive overview of existing approaches for deriving architectural models from requirement specifications. These approaches range from traditional methods to modern, model-based techniques, each addressing different facets of the transition process.

In [72], the authors address the challenges of transition from requirements engineering to software architecture by proposing a systematic method, the QuaDRA framework, aiming to bridge this gap with a focus on quality requirements. This approach ensures a more structured transition process, reducing reliance on individual expertise and facilitating broader adoption.

To improve the efficiency and accuracy of transitioning from requirements to architecture,

[74] introduces the use of artificial intelligence techniques. This method aims to semi-automatically generate architecture candidates, significantly reducing the manual effort typically required and improving productivity without compromising quality.

In a similar vein, [75] presents a comprehensive method for generating both individual and families of optimal software architectures. Through a real-world case study, the paper demonstrates how this approach optimizes multiple industry-grade products by balancing quality and effort, offering a practical framework for scalable and efficient architecture design.

Addressing the inherent ambiguity and inconsistency of natural language requirements, [76] proposes a tool-based systematic mapping technique. This approach translates requirements into components, facilitating the design of software architectures directly from user-defined inputs. By reducing ambiguity and improving traceability, this method ensures a smoother transition from informal to formal requirements specification.

The model-driven approach is further emphasized in [77]. This tutorial explores the hierarchical organization of requirements, viewing them as objects and modelling their associations. Integrating object-oriented principles enables a seamless transformation into high-level design models, considering the influence of non-functional requirements on architectural decisions.

The relationship between agile methodologies and architectural modelling is explored in [78]. This paper proposes a novel process that integrates patterns and components into requirements elicitation, aligning with agile principles. The approach balances the benefits of agile methods—such as risk reduction and customer involvement—with the explicit modelling of architecture to promote reuse and extend artifact applicability in future projects.

In [79], the authors present a process that incorporates architectural considerations into requirements modelling using the OpenUP method. This approach improves traceability and coherence between architecture and implementation, exemplified through its application in a service-oriented architecture project. Embedding architectural insights early in the development process ensures alignment between requirements and the final design.

The design of cyber-physical systems is another domain where bridging the gap between functional and architectural models is essential. [80] introduces the Assisted Transformation of Models (AST) method, which facilitates the transition from Simulink-based functional models to AADL architectural representations. By ensuring consistency and coupling between these models, AST provides a reliable foundation for designing systems like UAVs.

A goal-oriented perspective is adopted in [81]. This approach treats functional and non-functional requirements as goals to be achieved, refining them to derive logical architectures. By explicitly capturing the relationship between requirements and architectures, this method

enhances the ability to evaluate whether an architecture satisfies its requirements and supports better decision-making.

Lastly, [82] presents a methodology to link natural language requirements to architectural models. The approach eliminates ambiguity and establishes traceability links by employing restricted natural language and automated tools. This method is demonstrated through industrial case studies, showing its effectiveness in generating and refining AADL models from requirements.

Discussion

The literature reveals that textual specifications, particularly in natural language, continue to dominate as the primary input for transitioning requirements to architecture due to their accessibility and familiarity. However, their inherent ambiguity and lack of precision highlight the growing need for structured alternatives. Model-based approaches, including functional models, goal-oriented frameworks, and intermediate representations, offer promising methods to enhance traceability and reduce reliance on subjective interpretations. Tools aimed at automating component generation and improving requirement-to-architecture mappings have made strides in reducing manual effort and enhancing alignment across development phases, particularly in traceability. These innovations demonstrate a clear trend toward integrating architectural considerations earlier in the development lifecycle.

Despite these advances, significant limitations remain in addressing the unique demands of safety-critical systems. Many existing methods lack mechanisms to incorporate safety-related specifications and ensure compliance with aviation standards like DO-178C. The absence of dynamic adaptability in generated architectures and the limited focus on separating specification from design undermine their applicability in regulated domains. Furthermore, the reliance on human heuristics in tools and the insufficient attention to maintainability constrain their utility in evolving or highly regulated environments. Bridging these gaps requires approaches prioritizing safety-critical standards and flexibility, ensuring robust and reliable transitions from requirements to architecture.

The framework presented in Chapter 6 in this thesis ensures that the entire range of specified requirements, including non-functional and safety-related, are methodically incorporated into the architectural creation process, in contrast to many other approaches that mainly concentrate on functional requirements. In critical fields like aviation, where safety concerns need to be specifically taken into account at every step, this is a crucial aspect. Furthermore, another aspect that differentiates from other works is the user-friendly design, allowing the architecture to be generated directly within the same development tool as the source model.

This ensures a smooth, verifiable, and effective transition from requirements to architecture by removing the possibility of information loss that might happen when moving models between platforms. Additionally, given the expensive and time-consuming process involved in aviation, the time and cost saved with our contribution are important when compared to traditional architectural development methods, which include designing the architecture manually from the requirements.

2.6 Chapter Summary

This chapter provided a comprehensive foundation for the research by defining the essential background and concepts related to avionics software development. Important standards like ARINC 653 and DO-178C have been explained, emphasizing their importance in maintaining avionics systems' dependability and safety. Additionally, foundational principles of Formal Methods and MBE, including domain-specific modelling languages and model transformations, were introduced. Tools and languages critical to this research, such as AADL for architecture modelling and the LDRA Tool Suite for verification, were also discussed.

The chapter also included an extensive literature review that investigated software development and certification processes in avionics. It examined approaches integrating formal methods with automated tools to improve efficiency and rigour alongside methods for specifying HLRs in safety-critical systems. These approaches were analyzed with a focus on their alignment with DO-178C objectives, especially regarding how well they improved the capture and validation of safety-related elements. Additionally, the review included works on designing software architectures for safety-critical systems, emphasizing their application within the avionics sector.

CHAPTER 3 RESEARCH METHODOLOGY

This chapter presents the methodology to achieve the thesis objectives, outlining a workflow that integrates a literature review with three distinct contributions, as depicted in Figure 3.1. The thesis's structure, as shown in the figure, demonstrates how each contribution was generated with its primary stages and results following a survey of the literature, a study of the subjects addressed, and a definition of the research approach. The approach builds on foundational research and systematic development efforts, addressing safety-critical software development and certification challenges.

The initial step involved a comprehensive literature review aimed at understanding the context of avionics software development and certification. This review included an in-depth analysis of the DO-178C standard and its supplements to identify requirements imposed by certification authorities and mechanisms to ensure safety. It also explored existing methodologies, tools, and techniques for developing safety-critical systems, with particular emphasis on integrating formal methods and automated tools. These efforts were complemented by discussions with an industrial partner engaged in a UAV development project, enabling the identification of practical challenges and gaps in current practices. This collaborative effort informed the design of a novel methodology for software development, which integrates formal methods to enhance software robustness and automated tools to streamline processes and minimize errors.

The first contribution of this thesis is the development of a methodology tailored to software certification under DO-178C. This methodology provides a structured framework that defines key activities required to meet certification objectives while ensuring compliance with safety requirements. The integration of formal methods supports the verification of requirements, while automated tools address inefficiencies in manual tasks, aligning with industry needs. The methodology was validated through a case study, demonstrating its feasibility and effectiveness in a real-world context.

Building on insights gained from the literature review and the case study, the second contribution addresses the challenge of specifying HLRs with precision and traceability. The proposed solution is a domain-specific modelling language, ReDaML, designed to formalize the capture of essential requirements, particularly safety-related ones, and to ensure alignment with DO-178C objectives. ReDaML enhances the consistency and traceability of HLRs, reducing ambiguities and reinforcing the standard's guidelines. Its practicality was demonstrated through a case study that validated its application and utility in an industrial setting.

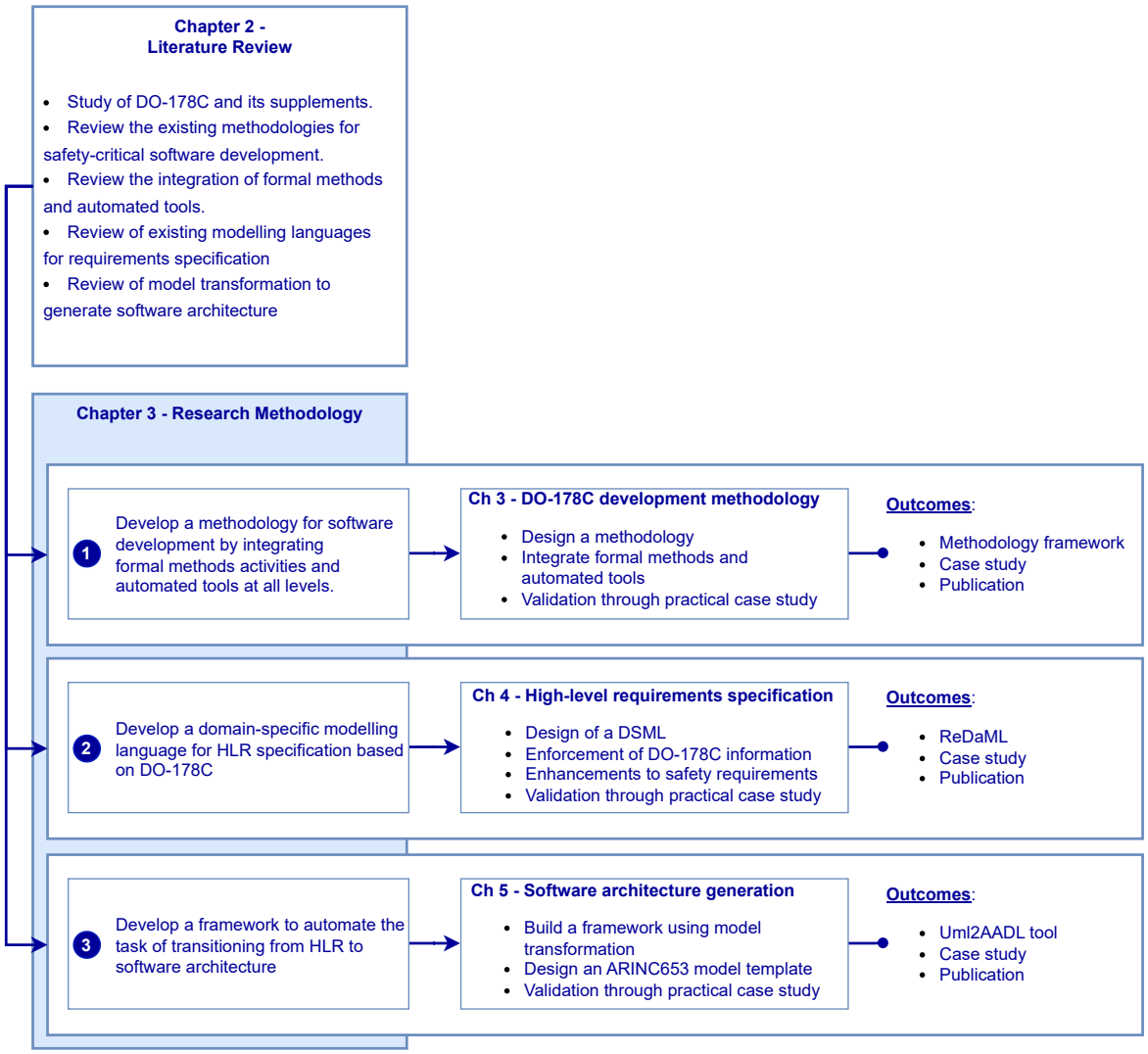

Figure 3.1 Research Methodology

The third contribution tackles the complexity of deriving software architecture from HLRs while ensuring full compliance with DO-178C. The case study revealed difficulties in maintaining traceability and addressing both functional and non-functional requirements during architecture design, particularly given the reliance on the expertise of software architects. To overcome these challenges, a framework was developed that employs model transformation mechanisms to generate software architectures automatically. This framework ensures traceability from HLRs, enforces compliance with the standard, and reduces reliance on manual processes, thereby improving consistency and quality.

Several challenges had to be overcome to construct this thesis, but each one added to the

insightful knowledge gained during the research process. One significant challenge was obtaining access to restricted documents and data, necessitating the use of alternate strategies to obtain pertinent data while maintaining confidentiality. Furthermore, because of its complicated regulatory framework and stringent certification requirements, comprehending the DO-178C standard proved to be a challenging undertaking. Determining appropriate tools based on the specific needs of the industrial partner presented another difficulty, requiring iterative adjustments to conform to realistic limitations and expectations. Additionally, there was a steep learning curve associated with adopting each contribution because it required mastery of many methodologies, modelling languages, and verification approaches. Overcoming these challenges strengthened the research contributions and their relevance in real-world situations by reaffirming the value of flexibility, organized problem-solving, and teamwork.

This thesis demonstrated a thorough awareness of the problems associated with designing critical software for certification. It is a meticulous process that necessitates particular attention to detail to ensure that every phase matches with the DO-178C objectives. Success in such projects necessitates collaboration among numerous stakeholders, each with a thorough understanding of the methodology to be used. Furthermore, the importance of modelling languages in critical system design was a great point. Compared to traditional, unstructured methods that rely primarily on natural language, formal modelling improves requirement clarity, decreases ambiguity, and strengthens the development process.

In summary, this chapter has described the methodology employed to achieve the research objectives, which include developing a certification-oriented methodology, a domain-specific modelling language for HLRs, and an automated framework for generating software architectures. These contributions collectively address critical gaps in safety-critical software development, advancing the state of the art while ensuring alignment with DO-178C and the evolving needs of the industry.

CHAPTER 4 A METHODOLOGY FOR DO-178C-COMPLIANT SOFTWARE DEVELOPMENT: INTEGRATING FORMAL METHODS AND AUTOMATED TOOLS

4.1 Chapter Overview

Developing software for UAVs used in safety-critical applications requires a careful balance between innovation and compliance with safety standards. Standards like DO-178C play a crucial role in this process by defining objectives that must be satisfied to ensure safety and reliability. Unlike prescriptive methodologies, DO-178C focuses on outcomes such as traceability, thorough verification, and compliance with safety requirements, offering flexibility in how these objectives are achieved. However, as highlighted in Chapter 2, this flexibility comes with challenges, including the complexity of ensuring complete traceability, managing extensive documentation and achieving robust verification in increasingly intricate UAV systems. Addressing these challenges requires a structured approach that aligns development practices with the goals of safety certification.

To tackle these issues, this study was carried out in collaboration with my doctoral colleague, Rim Zrelli, who made an equal contribution[1] to the proposed comprehensive methodology. This methodology integrates formal methods and automated tools, offering a systematic framework for developing DO-178C-compliant software. The methodology spans all four phases of the software lifecycle (requirements, design, coding, and integration), ensuring that the standard's defined objectives are consistently met. By focusing on traceability, early error detection, and streamlined documentation, the approach aims to mitigate the risks and inefficiencies often associated with developing safety-critical systems, particularly in the UAV domain.

Formal methods are a key component of this methodology, addressing the critical need for unambiguous specification and early validation. These techniques use mathematical models to define and verify system behaviour, ensuring clarity and precision in requirements and design. By identifying errors at early stages, formal methods reduce the likelihood of defects propagating through the lifecycle, ultimately improving software reliability and minimizing rework. Their inclusion ensures the software meets rigorous safety requirements while sup-

[1]Part of the content of this chapter is submitted for publication as: Zrelli, R.*, Misson, H. A.*, Kamkuimo, S., Ben Attia, M., de Magalhães, F. G., & Nicolescu, G. (2025). Integrating Formal Methods and Automated Tools for DO-178C Compliance in UAV Software. Submitted to ACM Transactions on Embedded Computing Systems. (*: Equal Contribution)

porting compliance with DO-178C objectives.

Automated tools complement formal methods by tackling the logistical challenges of certification, such as managing traceability and generating evidence for regulatory authorities. These tools provide efficient solutions for artifact management, continuous verification, and automated report generation, ensuring that documentation and certification evidence are accurate and complete. Together, the integration of formal methods and automated tools not only enhances the quality and reliability of UAV software but also provides a scalable and efficient framework for meeting the demands of safety-critical development.

4.2 Methodology

4.2.1 Proposed Methodology

The proposed methodology integrates formal methods with automated verification tools to ensure a comprehensive, traceable, and certifiable software development process for UAV systems, adhering to the rigorous standards set by DO-178C. The methodology's primary objective is to systematically address the key challenges associated with developing safety-critical software for UAVs by ensuring that each phase of the software development lifecycle—from requirements through integration—meets DO-178C compliance requirements.

The core of this methodology lies in combining the mathematical rigour of formal methods with the efficiency and automation provided by specialized verification tools. This hybrid approach ensures that each phase of the development process is both verifiable and traceable, producing the necessary certification artifacts required by aviation authorities. The methodology systematically spans the requirements, design, coding, and integration phases, ensuring comprehensive coverage of DO-178C objectives from the initial specification to the final system integration.

Formal Methods for Rigorous Specification and Verification

The use of formal methods in this methodology ensures that all system behaviours are specified with mathematical precision, reducing the likelihood of ambiguity in the requirements and design phases. Formal methods are applied to define and verify key aspects of the system through the following techniques:

- Formal Specification: System requirements (SRATS) and software requirements (HLRs and LLRs), are modelled using formal languages such as Alloy. These formal models enable the precise expression of system properties, eliminating ambiguities that could

lead to misinterpretations during implementation and ensuring consistency between the specifications.

- Formal Verification: The methodology employs model-checking techniques to verify the correctness of software application against the specified requirements. Tools such as SPIN and ESBMC are used to perform exhaustive exploration of the system's state space, ensuring that the specified properties hold across all possible execution paths. This is particularly valuable for detecting deadlocks, invalid states, coding errors and violations and unanticipated interactions between subsystems at an early stage of development.

The formal methods employed ensure that the critical components of the UAV software are verified not only against functional requirements but also against safety properties, such as freedom from deadlock, and ensuring correctness during the requirements and design phases which are essential for compliance with DO-178C.

Automated Tools for Continuous Verification and Traceability

To complement formal methods, the methodology employs automated verification tools, specifically the LDRA tool suite, which provides capabilities for managing requirements, performing code static and dynamic analysis, and ensuring end-to-end traceability and compliance with standard objectives. The use of qualified tools ensures that the entire development process—from the initial requirements specification through to the final integration of the system—is continuously monitored for compliance with DO-178C objectives. By automating much of the verification and validation work, this tool suite reduces the risk of human error, enhances traceability, and accelerates the generation of certification artifacts.

Key contributions of automated tools in this methodology include:

- Automated Traceability: Traceability across the software lifecycle is created for each phase, ensuring bidirectional links between requirements, design, code, and tests. This traceability is critical for demonstrating compliance with DO-178C, where each software artifact must be traceable to its originating requirement. The use of automated traceability ensures that changes in requirements are automatically reflected in related design and code elements, thereby maintaining consistency throughout the development process.

- Static and Dynamic Analysis: Automated verification using static and dynamic analysis tools ensures that code adheres to coding standards and meets the design requirements.

This is achieved through automated code reviews, dynamic testing, and unit testing, providing continuous feedback on the quality and compliance of the software.

- Certification Evidence Generation: The LDRA tool suite automates the generation of certification evidence, such as Software Accomplishment Summaries (SAS), traceability matrices, and verification reports, which are essential for DO-178C compliance. This automation accelerates the certification process and reduces the risk of human error in preparing certification artifacts.

4.2.2 Formal Methods and Automated Tools: A Unified Approach

A key innovation of this methodology is its seamless integration of formal methods and automated tools. While formal methods provide the mathematical foundation for specifying and verifying system behaviours, automated tools offer practical solutions for managing traceability and continuously validating system components. The proposed methodology thus combines the rigour of formal verification with the efficiency of automation, ensuring that all software components are developed and tested in accordance with DO-178C.

The combination of these two approaches ensures that each phase of the software development lifecycle is covered:

- Requirements Phase: Formal methods are employed to specify HLRs, which ensures accuracy and precision as well as conformity with SRATS. Meanwhile, automation seeks to maintain traceability and conformance to standards.

- Design Phase: Formalization also aims to ensure the consistency of LLRs and architecture, and formal verification, using Model Cheking, is used to ensure that these are compliant with their high-level counterparts. Automation aims to help with the traceability and verification of artifacts.

- Coding Phase: Automated static and dynamic analysis tools provide continuous feedback on code quality and ensure adherence to the design.

- Integration Phase: Automated testing tools verify that all subsystems interact correctly, producing certification evidence required for DO-178C compliance.

This methodology is designed not only for rigorous verification but also for scalability. The combination of formal methods and automation allows the process to be efficiently applied to complex UAV systems, where manual verification would be impractical. Furthermore, the automation provided by the tool streamlines many of the time-consuming aspects of

verification, making the methodology scalable to larger projects without sacrificing the rigour needed for DO-178C compliance.

4.3 Process Description

This section outlines a detailed step-by-step approach to transforming system requirements into certified software, adhering strictly to the DO-178C objectives. The process spans all phases of the software development lifecycle: Requirements, Design, Coding, and Integration. For each phase, the methodology employs formal methods and automated verification tools, leveraging the LDRA Tool Suite to ensure compliance, traceability, and verification.

4.3.1 Software Requirements Phase

The Software Requirements Phase serves as the foundational step in developing DO-178C-compliant software, focusing on capturing and managing precise, unambiguous software high-level requirements. This phase aims to ensure that all functional and non-functional requirements are well-defined, verified, and traceable, forming a reliable basis for subsequent development stages. Meeting DO-178C objectives such as compliance, traceability, accuracy, consistency, and verifiability (objectives A3.1, A3.6, A3.2, A3.4) is paramount throughout this phase.

Figure 4.1 illustrates the entire Software Requirements Phase, showing the sequence from requirements collection to formal specification, analysis, and verification. Each step is depicted with key outputs that feed into the next stage of development, emphasizing the bidirectional traceability and formal verification processes. This diagram highlights the integration of formal methods and automated tools, ensuring that the process remains aligned with DO-178C objectives throughout.

Requirements Configuration and Development The first step of the process consists of collecting and organizing the system's functional and non-functional requirements. This activity involves documenting the system's behaviour, performance, and constraints to ensure all aspects of the system are adequately covered. The configuration of requirements must enable proper traceability and refinement in subsequent phases. Clear and thorough requirements documentation ensures the system's intended functionality is well-defined from the outset.

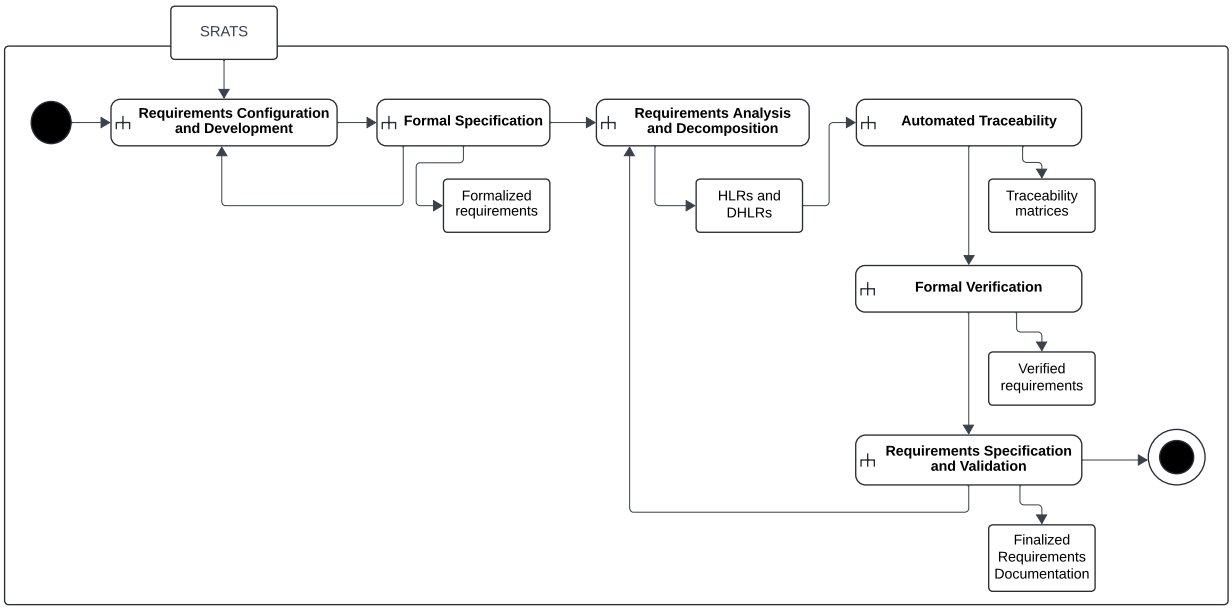

Figure 4.1 Expanded view of the software requirements process.

Formal Specification In this step, the requirements are formalized to eliminate ambiguities and ensure a mathematically sound foundation for the HLR that will be developed. Formal specification provides a precise and verifiable model of the requirements, reducing the risks of misinterpretation and errors in later stages of development. Formal specification enables the system behaviour to be represented unambiguously, ensuring it aligns with the project's functional and safety goals. Formal specification methods like Alloy, Z notation or B method can be used to specify requirements using a well-defined syntax and semantics.

Requirements Analysis and Decomposition This activity involves analyzing the SRATS and breaking them down into more detailed, manageable components, such as HLRs and DHLRs. This decomposition helps to clarify system functionalities, making it easier to design, implement, and test. Possible derived requirements may be required to supplement the expected behaviour from a software perspective. If it has been developed, it should be sent back to the system team so that they can determine if no impact was added to the system.

Automated Traceability Automated traceability ensures that all system and high-level requirements (SRATS and HLRs) are traceable through subsequent phases of the software lifecycle, including design, coding, and testing. Establishing bidirectional links between requirements and future development artifacts ensures that each requirement is properly implemented and verified, preventing inconsistencies or overlooked requirements as the project

progresses. Maintaining traceability early in the lifecycle is crucial for demonstrating compliance with DO-178C and ensuring comprehensive coverage.

This activity focuses on continuously linking requirements to artifacts, ensuring that any changes made during the requirements phase are reflected across all relevant documents and models. In turn, this establishes a robust foundation for traceability in later phases, when the design and testing will build on these requirements.

Formal Verification Early in the lifecycle, it is important to verify that the requirements are correct and complete, as well as to detect and resolve any potential conflicts or errors before they propagate to later stages. Formal verification techniques, such as model checking or theorem proving, are applied to ensure the system's properties meet their intended specification. This helps detect inconsistencies, deadlocks, or invalid states.

Model-checking techniques can be used to verify that the specified requirements are accurate and consistent with system behaviour. This ensured compliance with DO-178C objectives A3.2 (accuracy) and A3.4 (verifiability). The early application of formal verification allowed us to detect potential errors before they impacted later development phases.

Requirements Specification and Validation The final activity in this phase involves specifying the detailed requirements in a form that is consistent, complete, and verifiable. This includes reviewing the requirements documentation, refining the specification where necessary, and validating it to ensure alignment with both system goals and regulatory standards. The validated requirements are then used as the basis for the subsequent design phase.

Peer reviews and formal validation techniques must be applied to ensure that the requirements are accurate and complete. Model checking can also be used to validate high-level requirements, ensuring alignment with DO-178C objectives A3.2 (accuracy and consistency), A3.4 (verifiability) and A3.5 (conformity to standards). This helps ensure that all system requirements are covered.

For a quick reference, Table 4.1 summarizes the key activities, the tools used, the outputs generated, and the DO-178C objectives addressed during the Software Requirements Phase.

4.3.2 Software Design Phase

The Software Design Phase is a critical stage where the software application architecture and detailed design are defined, ensuring that the software is designed to match all of the

Table 4.1 Summary of Activities, Outputs, and DO-178C Objectives Addressed in the Software Requirements Phase

Activity	**Output**	**DO-178C Objective Addressed**
Requirements Configuration and Development	Comprehensive system requirements	
Formal Specification	Formalized, unambiguous requirements	
Requirements Analysis and Decomposition	Detailed HLRs, DHLRs	A2.1 (Developed HLRs), A2.2 (Defined DHLRs)
Automated Traceability	Traceability matrices	A3.1 (Compliance), A3.6 (Traceability)
Formal Verification	Verified requirements, validation reports	A3.2 (Accuracy), A3.4 (Verifiability), A3.5 (Conformity to standards)
Requirements Specification and validation	Validated requirements documentation	A3.2 (Accuracy), A3.4 (Verifiability), A3.5 (Conformity to standards)

requirements gathered in the preceding phase and is detailed enough to assist developers in writing code. This phase ensures that the software architecture is systematically designed and that LLRs are created and traceable to HLRs. The goal is to ensure that the software design is robust, modular, and verifiable, complying with DO-178C objectives. The phase also incorporates formal verification techniques to ensure logical consistency and safety in the design.

Figure 4.2 illustrates the entire Software Design Phase, showing the sequence from subsystem identification to design verification and review. Each step is depicted with key outputs that feed into the subsequent development stages, emphasizing the traceability and formal verification processes.

Subsystem Identification and Decomposition This activity involves breaking down the system into smaller, manageable subsystems, facilitating easier implementation and verification. The purpose is to modularize the system, making it easier to design, implement, and test individual components without compromising the overall architecture. Subsystem identification provides the foundation for establishing clear boundaries and responsibilities for each subsystem, ensuring modularity and scalability.

In our case study, the decomposition produced subsystem models and diagrams, establishing a well-defined structure for the overall system architecture.

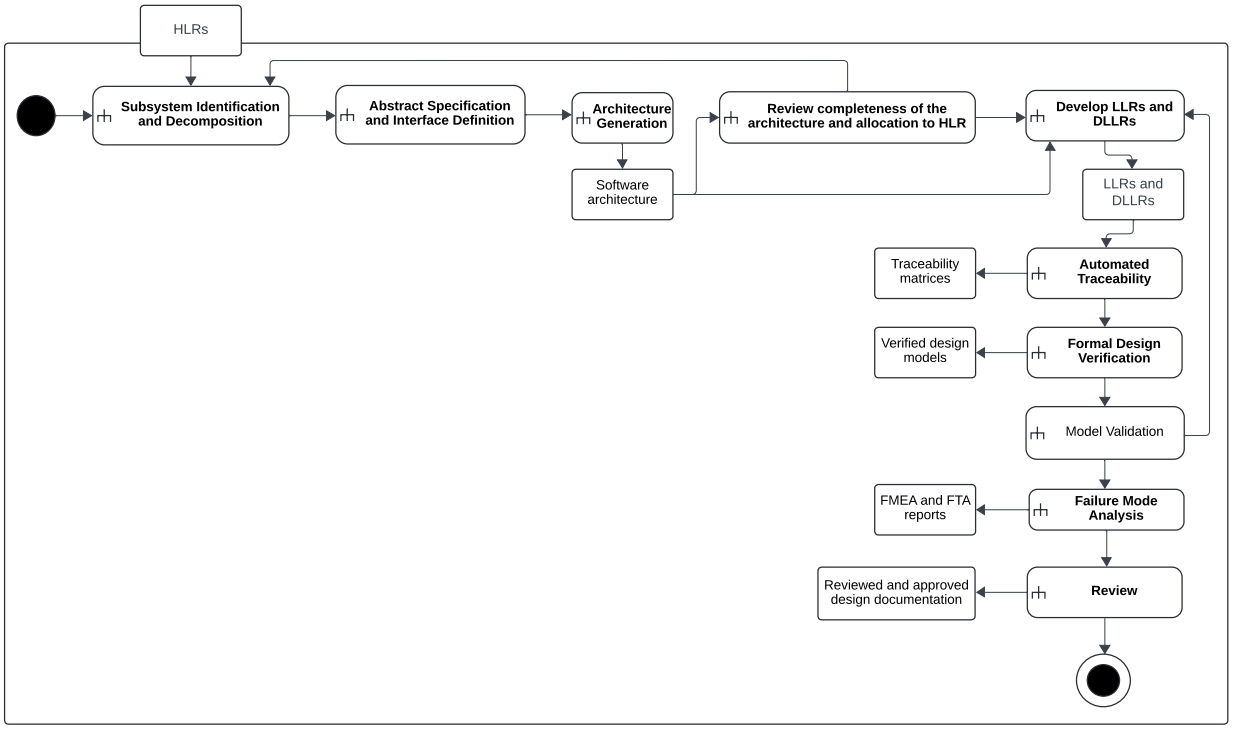

Figure 4.2 Expanded view of the software design process.

Abstract Specification and Interface Definition The abstract specification focuses on determining the detailed structure of each subsystem identified in the previous activity. This includes outlining how each subsystem will function, as well as specifying inputs and outputs that direct component interactions. The task also includes producing detailed interface documentation that captures both control and data flow across subsystems. This documentation serves as a blueprint for the system's interactions, ensuring that the behaviour of each subsystem is consistent with the overall system design. By precisely specifying these interfaces, this step promotes consistency, simplifies integration, and establishes the framework for eventual implementation and verification.

Architecture Generation The Architecture Generation activity is responsible for building the structure of the software application, with a focus on describing and displaying subsystem interactions using thorough diagrams. The architecture should be documented in a consistent and easily accessible design so that developers can use it as a foundation for both implementation and maintenance. Clear and well-organized documentation enables future developers to efficiently maintain and expand the software architecture, which is especially important for long-term projects.

Review Completeness of the Architecture and Allocation to HLR Once the architecture is generated, it is essential to review the architecture's completeness and ensure it aligns with the HLRs. Following DO-178C specifications, the architecture must be linked with HLRs, ensuring a clear mapping or traceability between them. This ensures that each architectural component is justified and tied to specific requirements, which aids certification efforts. The review ensures that no critical elements are omitted and that the architecture is well-suited to address the needs outlined by the HLRs.

LLR and DLLR Development This activity focuses on converting HLRs into comprehensive specifications that will guide software implementation. The LLRs give the amount of detail required for developers to accurately produce the software by defining functional behaviour, algorithms, and constraints for each component. The LLRs also include a data dictionary, which defines data types, variables, and structures, ensuring that data is handled consistently across the code. In cases where additional behavioural details are needed to support or clarify the primary requirements, DLLRs are created.

By producing precise and comprehensive LLRs and DLLRs, this process assures that the software meets both design expectations and system requirements.

Automated Traceability Similar to the requirements phase, maintaining traceability between design elements and requirements is critical. Automated traceability ensures that all design components are linked to their respective HLRs, LLRs, and other development artifacts. This guarantees that the design accurately reflects the system's requirements and that any changes to the design can be traced back to their originating requirement.

Formal Design Verification This activity seeks to confirm the accuracy, consistency, and completeness of LLRs and DLLRs, if applicable, through a rigorous verification procedure. This activity ensures that the LLRs appropriately reflect the HLRs' intent and that all system behaviours are well-defined and free of ambiguities. Model-checking techniques are frequently used to perform formal verification, giving a mathematical foundation for determining conformity between LLRs and HLRs.

This technique aids with the early detection of inconsistencies, unwanted actions, and potential design faults by investigating all conceivable states and behaviours within the design. This level of formal verification improves the reliability of the software design and increases confidence that the implementation will precisely meet the requirements.

Model Validation Before proceeding to the coding phase, it is essential to validate the design models to ensure they accurately reflect the requirements. This step aims to confirm that the LLRs completely capture the intended functionality and accurately represent the HLRs. This activity involves validating the LLRs by reviewing the verification results to ensure consistency and completeness. During Model Validation, any discrepancies, missing requirements, or omitted behaviours cause the LLRs and DLLRs to be re-evaluated (step back to "Develop LLRs and DLLRs activity"). Corrections are then made to ensure that the model reflects the original system intent. This activity ensures a solid basis for code implementation by carefully validating the model representing low-level requirements, reducing the chance of issues occurring later in the development process.

Failure Mode Analysis The process identifies and mitigates potential failure modes in the software design to ensure overall system safety and dependability. This procedure typically involves performing Failure Mode and Effects Analysis (FMEA) and Fault Tree Analysis. FMEA thoroughly investigates each component and function to identify possible failure modes, causes, and effects, allowing the design team to prioritize and fix important vulnerabilities. FTA, on the other hand, examines how a series of failures can lead to a critical or catastrophic event, outlining dependencies and likely root causes in a systematic, hierarchical manner. Together, FMEA and FTA provide a comprehensive perspective of potential failure situations, guaranteeing that the software design meets the necessary safety requirements.

Review The final activity in the design phase involves reviewing the design to ensure it meets all quality and safety standards. This step ensures that the design is accurate, complete, and ready for implementation. Reviews help identify any potential design flaws or areas that require further refinement.

For a concise summary, Table 4.2 highlights the key activities, tools used, outputs generated, and the DO-178C objectives addressed during this phase.

4.3.3 Software Coding Phase

The Software Coding process involves translating design specifications into source code that meets high safety and reliability standards for safety-critical systems. In this phase, developers implement code based on the comprehensive descriptions created throughout the design stages, guaranteeing strict compliance with DO-178C criteria. The coding phase emphasizes traceability, allowing each line of code to be linked back to its appropriate LLRs and, as a result, HLRs, thereby confirming compliance with the system's intended functionality and

Table 4.2 Summary of Activities, Outputs, and DO-178C Objectives Addressed in the Software Design Phase

Activity	**Output**	**DO-178C Objective Addressed**
Subsystem Identification and Decomposition	Subsystem models, decomposition diagrams	
Abstract Specification and Interface Definition	Interface definitions, abstract specifications	
Architecture Generation	Software architecture models	A2.3 (Architecture development)
Review completeness of the architecture and allocation to HLR	Verified architecture allocated to HLRs	A4.8 (Architecture's compatibility with HLRs), A4.9 (Architecture's consistency) , A4.11 (Architecture's verifiability)
Develop LLRs and DLLRs	Documented LLRs and DLLRs	A2.4 (Developed LLRs), A2.5 (Defined DLLRs)
Automated Traceability	Traceability matrices	A4.6 (Traceability)
Formal Design Verification	Verified design models	A4.1 (LLRs compliance with HLRs), A4.2 (LLRs accuracy), A4.4 (LLRs verifiability)
Model Validation	Validated design models	A4.1 (LLRs compliance), A4.2 (LLRs accuracy)
Failure Mode Analysis	FMEA and FTA reports	
Review	Reviewed and approved design documentation	A4.1 (LLRs' compliance with HLRs), A4.5 (LLRs' conformity to standards), A4.8 (Architecture's compatibility with HLRs), A4.9 (Architecture's consistency), A4.12 (Architecture's conformity to standards).

safety objectives.

To maintain high quality, automated tools and formal methods are used to check code consistency, find errors, and guarantee coding standards are followed. This approach guarantees that the final code is robust, verifiable, and perfectly aligned with the stated architecture and requirements.

Figure 4.3 provides a visual representation of the Software Coding Phase. It illustrates the sequential flow from code generation to formal verification and testing, emphasizing the integration of automated analysis and validation processes.

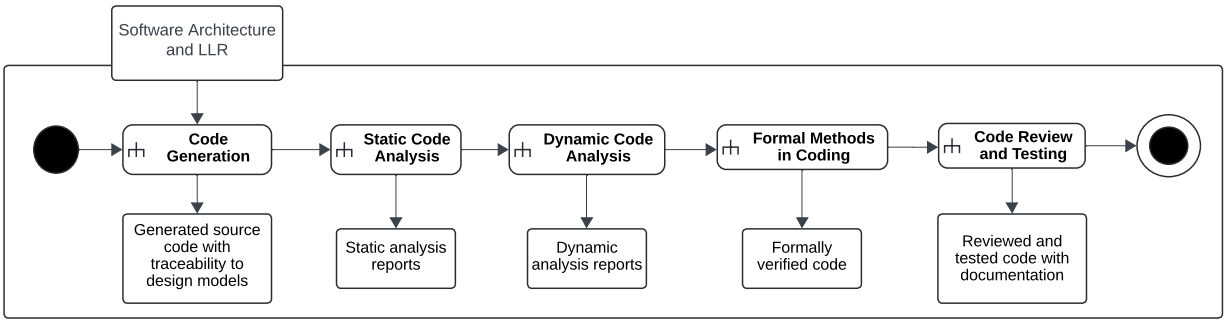

Figure 4.3 Expanded view of the software coding process.

Code Generation This activity focuses on generating source code directly from design models, ensuring that the implementation adheres precisely to the specified design and functional requirements. The fundamental goal of this step is to preserve consistency and traceability so that each code component can be easily connected back to its appropriate design elements. Code can be generated using qualified tools that automate the process and reduce human error, or it can be written manually. A hybrid technique can also be used, combining automatic generation and manual development. This activity lays the basis for accurate, reliable code, assisting the certification process by meeting DO-178C traceability and quality standards.

Static Code Analysis Static code analysis is a key activity in the coding process that checks source code without running it, revealing potential flaws early in development. This analysis identifies code standard violations, data flow inconsistencies, and other quality issues, allowing developers to correct them before they affect runtime behaviour. This activity helps to create reliable, high-quality software that is easier to maintain and more compliant with DO-178C regulations by verifying the code adheres to set safety and quality criteria.

Dynamic Code Analysis Dynamic code analysis examines the code's behaviour during execution, ensuring that it works properly under a variety of scenarios and satisfies performance and safety expectations. This activity entails running the software and observing its actual reactions, which include looking for runtime faults, timing difficulties, and memory utilization. Code coverage is an important aspect of dynamic analysis since it ensures that all pathways and circumstances within the code are checked, allowing us to ensure that no functionality is left untested. Dynamic code analysis, which simulates real-world events, certifies the code's suitability for deployment in a safety-critical environment, hence supporting DO-178C's emphasis on dependable and verifiable software behaviour.

Formal Methods in Coding Formal Methods in Coding add mathematical rigour to the software verification process, allowing for the exact detection of potential errors within the code. Using formal verification approaches, such as bounded model checking, developers can systematically investigate a finite subset of the software execution across all potential inputs. This method enables thorough verification of critical properties, such as correctness, safety, and conformance to design requirements. Bounded model checking, in particular, can reveal subtle issues that typical testing approaches may miss by investigating edge cases and unexpected input possibilities in a controlled, repeatable manner. Through formal approaches, this activity provides an additional degree of confidence in code reliability and conformance with DO-178C standards, ensuring that the software is error-free and behaves as intended before moving into the testing and final verification stages.

Code Review and Testing The Code Review and Testing activity is the final step in the coding process, and it involves a thorough examination to ensure that the code meets all quality standards and functional requirements. This step includes performing systematic code reviews to ensure that the implementation appropriately reflects the planned functionality. Following the review, testing is carried out to check for any remaining errors and ensure that the software functions properly under expected settings.

4.3.4 Software Integration Phase

The Software Integration Phase is the final step in the development lifecycle, where all software components are integrated, tested, and verified to ensure they work together as intended. This phase aims to validate that the compiled and integrated code meets the system requirements, functions correctly in an integrated environment, and is ready for deployment. The activities in this phase ensure compliance with key DO-178C objectives related to integration,

verification, and validation.

Source Code Analysis and Compilation The first step in the integration phase is to analyze and compile the source code. This ensures that all individual modules are correctly compiled and linked to form executable code without any errors. Compilation checks help identify any issues related to code compatibility, linking errors, or violations of system specifications.

During this activity, industry-standard compilation and linking procedures are used to verify the integrity of the source code, ensuring that it produces a valid executable. This step aligns with DO-178C objective A2.7, which focuses on ensuring that the software is compiled and linked correctly to maintain integrity throughout the integration process.

Automated Integration Testing Once the software components are compiled, they are integrated and tested to ensure that the different modules interact correctly. The goal of automated integration testing is to validate that all components function together seamlessly and meet the overall system requirements.

Automated integration testing verifies that the integrated modules perform as expected when operating as part of a unified system. This activity helps detect any integration issues, such as incorrect data handling between modules, timing errors, or unexpected interactions. By performing comprehensive testing of module interactions, this activity addresses DO-178C objective A5.7, which focuses on ensuring that the software integration process is complete and correct.

End-to-End Verification It ensures that the entire system meets its specified requirements and functions correctly when deployed in a fully integrated environment. This step involves executing comprehensive tests that validate the system's performance, reliability, and compliance with requirements.

The end-to-end verification process ensures that the integrated software is free from defects, performs as expected, and adheres to the system requirements. It involves rigorous testing across different scenarios to ensure that the system meets DO-178C objectives A6.1 (compliance with HLRs), A6.2 (robustness with HLRs), A6.3 (compliance with LLRs), and A6.4 (robustness with LLRs).

Parameter Data Items (PDI) Generation As the final step in the integration phase, PDIs are generated. PDIs refer to configuration files and system parameters that ensure the

correct deployment and operation of the software in its target environment. The generation of PDIs is critical for configuring the system correctly and ensuring that all necessary data items are included for proper functionality.

This activity produces the necessary PDIs to configure the system for operation, ensuring that the software is tailored to its intended use environment. The generation of PDIs (DO-178C objective A5.8) is an essential part of delivering a complete, deployable system.

The Software Integration Phase, as depicted in Figure 4.4 ensures that all software components are properly compiled, tested, and verified before deployment. Each step in this phase—from code analysis and compilation to end-to-end verification—ensures that the system operates as a unified whole and meets the specified requirements.

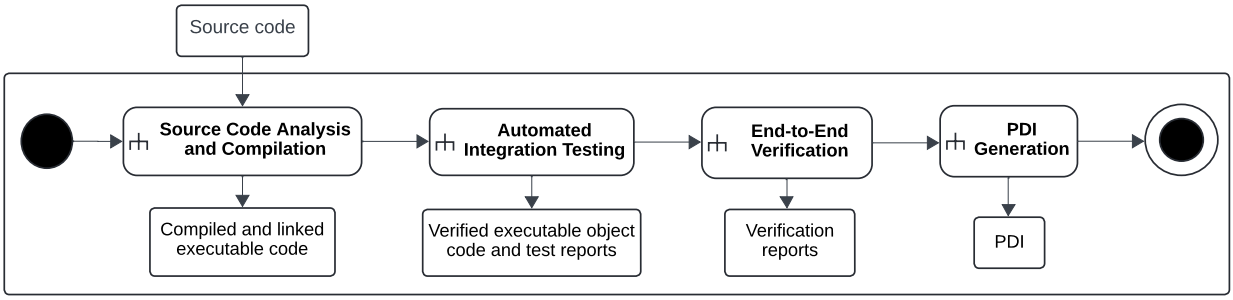

Figure 4.4 Expanded view of the software integration process.

4.4 Case Study: Collision Avoidance System for UAVs

4.4.1 Overview of Collision Avoidance System

The Collision Avoidance System (CAS) for UAVs is a crucial technology designed to ensure the safety and reliability of autonomous UAV operations by enabling the detection, evaluation, and avoidance of potential collision threats in real-time. As UAVs become more prevalent in both commercial and defence sectors, safely integrating them into increasingly congested airspaces is paramount. The CAS addresses this challenge by autonomously replicating and enhancing traditional "see-and-avoid" techniques used by human pilots, enabling UAVs to operate safely in dynamic airspace environments.

The CAS continuously monitors the UAV's surroundings, identifying both cooperative (transponder-equipped) and non-cooperative aircraft or objects. It plays a particularly vital role in Beyond Visual Line of Sight (BVLOS) operations, where human operators are unable to maintain direct visual contact with the UAV. In such cases, the CAS autonomously assesses colli-

sion risks, prioritizes threats, and determines appropriate avoidance maneuvers to ensure the UAV maintains a safe flight path. This capability is crucial for UAVs operating in airspace shared with manned aircraft, where real-time, data-driven decision-making is required to avoid mid-air collisions.

As a Design Assurance Level B (DAL-B) system under the DO-178C standard, the CAS is classified as critical to preventing hazardous situations, demanding robust software development, verification, and validation processes. The DAL-B classification emphasizes the system's importance in ensuring reliable performance under all operational conditions, as any failure could lead to catastrophic consequences.

The key functions of the CAS include:

- Traffic Detection: The system continuously monitors the UAV's surroundings, detecting both cooperative and non-cooperative aircraft or objects.
- Track Traffic: The previous traffic detected is tracked to establish a reliable track history for each object.
- Collision Risk Assessment: Once traffic is detected, the CAS evaluates the potential for a collision based on trajectory predictions, considering speed, direction, and proximity.
- Threat Prioritization: In situations where multiple collision threats are identified, the system prioritizes these threats based on proximity and the immediacy of potential impact.
- Avoidance Maneuver Determination: CAS autonomously determines the most appropriate avoidance maneuver (e.g., altitude adjustment or lateral deviation) to prevent a collision with the highest-priority threat.
- Maneuver Command: The system sends maneuver commands to the UAV's flight control system to execute the avoidance action in real-time.

Inspired by NASA's advancements in non-cooperative collision avoidance [83], the CAS implementation seeks to enhance UAVs' ability to autonomously navigate congested airspaces with minimal human intervention. This system aligns with efforts to establish higher safety standards for UAVs in civil airspace, ensuring they meet regulatory safety requirements, particularly as outlined in DO-178C. By autonomously detecting, evaluating, and avoiding collisions, CAS serves as a critical enabler for UAV operations, facilitating their safe and reliable integration into future airspace systems.

4.4.2 Implementation of Methodology

The proposed methodology was applied to the development and verification of the CAS for UAVs, focusing on the key phases of Requirements, Design, and Coding to ensure compliance with DO-178C. Given the system's critical DAL-B classification, each phase was executed with rigour, integrating formal methods and automated tools to ensure traceability, verifiability, and safety.

The implementation of the methodology was tailored to the specific needs of the CAS development, leveraging tools that aligned with the project's objectives and constraints. While specific tools, such as the LDRA tool suite, were employed for this case study, the methodology itself is designed to be adaptable, enabling the use of alternative tools and platforms that meet project-specific requirements. This section highlights the functionality of the tools used in this implementation while underscoring the broader applicability of the proposed approach. Each phase was iteratively refined based on model-checking results, verification activities, and updates to system requirements, ensuring that the CAS development was robust and aligned with certification standards.

Requirements Phase

The first phase of the methodology focuses on defining and evaluating the CAS requirements for DO-178C compliance. This entailed identifying the system-level requirements (SRATS), including functional and non-functional aspects critical to CAS performance and safety. To ensure these requirements were clear, unambiguous, and precise, a formal model of the SRATS was developed using the Alloy language, and the Alloy Analyzer was used to conduct rigorous consistency checks.

A notable example is SRATS_002, which specifies the spatial boundaries of CAS surveillance based on detection range, azimuth, and elevation fields of regard. The Alloy model for SRATS_002 is as follows:

```
    fact SRATS_002 {
    all sys: CollisionAvoidanceSystem |
        sys.surveillanceVolume.traffic = { t: Traffic |
            t.x <= sys.detectionRange and
            (t.y >= sys.minAzimuthFieldOfRegard and t.y <= sys.azimuthFieldOfRegard) and
            (t.z >= sys.minElevationFieldOfRegard and t.z <= sys.elevationFieldOfRegard)
        }
}
```

Listing 4.1 Alloy Specification for SRATS_002

This Alloy model formally encapsulates the spatial constraints of the CAS surveillance vol-

ume. Assertions, such as *CompleteTrafficDetection*, were used to verify that all traffic meeting these criteria was correctly detected. The Alloy Analyzer validated this assertion without counterexample, confirming the logical consistency of the system model under the tested scenarios.

During the verification process of other assertions, Alloy Analyzer discovered several inconsistencies in the SRATS, revealing issues such as incorrect assumptions about traffic detection range and ambiguities in threat prioritization. These findings were documented as defects and reported to the system engineering team for correction. Each issue was addressed using an iterative refinement strategy, resulting in a more accurate and reliable set of system requirements consistent with the desired system behaviour and safety objectives. A comprehensive list of SRATS is provided in Appendix A for reference.

Following the refinement and verification of the SRATS, the validated requirements were imported into LDRA TBManager to support the HLR development. The SRATS were systematically decomposed into key CAS functions, enabling the creation of software-level HLRs (detailed in Appendix A) that directly traced back to the system requirements.

The initial draft of the HLRs was developed and set up within a versioned baseline in the tool. As depicted in Figure 4.5, a traceability was created indicating that each HLR was directly linked to its associated SRATS. Following the outlined development activities, the next stage was to ensure that the requirements fulfilled DO-178C objectives, specifically those listed in *Table A-3 ("Verification of Outputs of Software Requirement Process")* of the DO-178C.

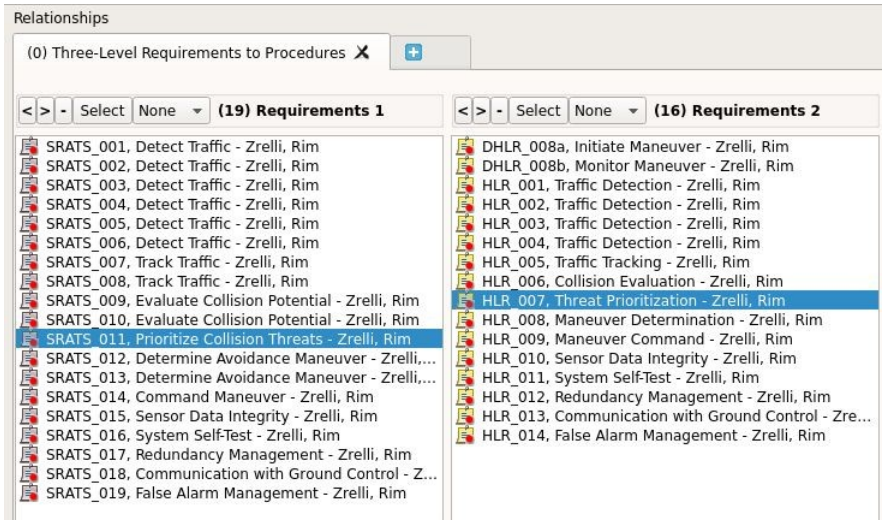

Figure 4.5 TBmanager relationship view between SRATS and HLR

To ensure that the HLRs adhered to DO-178C objectives for compliance with SRATS, ac-

curacy, consistency, and verifiability, a formal model was constructed in Alloy. This model enabled the detection of inconsistencies and ambiguities, facilitating early refinement and validation of the requirements.

The verification process involved:

- Initial Model Execution: The initial Alloy execution demonstrated that the HLRs were logically consistent and free of major contradictions. However, one critical assertion, *InadequateCollisionThreatHandling*, failed, exposing gaps in HLR_008 and HLR_009, which relate to the handling of collision threats and associated maneuvers.
- Analysis of the counterexample: The failed assertion revealed a scenario where identified collision threats did not trigger corresponding avoidance maneuvers. This gap was traced to ambiguities in the activation and prioritization conditions for maneuvers:
  - Ambiguity in maneuver activation: HLR_008 lacked specificity about when maneuvers should be initiated for detected threats.
  - Gaps in prioritization logic: HLR_009 did not define how to prioritize maneuvers when multiple threats were present.
  - Dynamic adjustment issues: Conditions for dynamically updating or terminating maneuvers were insufficiently detailed.
- Refinement: Based on this analysis, HLR_008 and HLR_009 were refined to explicitly mandate the initiation of maneuvers for all detected threats unless overridden by a higher-priority threat and to address maneuver termination and prioritization. To bridge the gap between HLRs and LLRs, two new DHLRs (DHLR_008a and DHLR_008b) and their rationale were introduced to clarify the conditions for handling collision threats.
- Reverification: The refined HLRs and DHLRs were re-modelled in Alloy. Assertions, including *InadequateCollisionThreatHandling*, were verified (as depicted in Figure 4.6), and no further counterexamples were found. This confirmed compliance with DO-178C objectives A3.2 (accuracy), A3.4 (verifiability), and A3.5 (conformity to standards).

The assertion *InadequateCollisionThreatHandling* states that some traffic detected as a threat is not being actively maneuvered against when it should be and it is specified as follows:

```
assert InadequateCollisionThreatHandling{
    all sys: CollisionAvoidanceSystem, t: Traffic |
        t in sys.collisionThreats implies (
```

```
Executing "Check InadequateCollisionThreatHandling for 5"
  Solver=sat4j Bitwidth=4 MaxSeq=5 SkolemDepth=1 Symmetry=20 Mode=batch
  21596 vars. 1730 primary vars. 34258 clauses. 864ms.
  No counterexample found. Assertion may be valid. 29ms.
```

Figure 4.6 Results for checking the InadequateCollisionThreatHandling assertion after refinement

```
            some m: sys.maneuvers |
            m.isActive = True and
            t in m.threat.spaceZone and
            (no t2: sys.collisionThreats - t | t2.threatLevel > t.
               threatLevel and t2.timeToCollision < t.
               timeToCollision)
        )
}
check InadequateCollisionThreatHandling for 5
```

Listing 4.2 InadequateCollisionThreatHandling Assertion

The validated HLRs were imported back into TBManager to finalize the traceability matrix and link each HLR and DHLR to its corresponding SRATS. This integration facilitated the generation of certification artifacts, such as the Software Requirements Traceability Matrix (SRTM) (Figure 4.7).

Requireme...	DHLR_008a, I...	DHLR_008b, ...	HLR_001, Tra...	HLR_0010, S...	HLR_0011, S...	HLR_0012, R...	HLR_002, Tra...	HLR_003, Tra...	HLR_004, Tra...	HLR_005, Tra...	HLR_006, Col...	HLR_007, Thr...	HLR_008, Ma...	HLR_009, Ma...	HLR_013, Co...	HLR_014, Fal...
SRATS_001			X													
SRATS_002			X													
SRATS_003							X									
SRATS_004								X								
SRATS_005								X								
SRATS_006									X							
SRATS_007										X						
SRATS_008										X						
SRATS_009											X					
SRATS_010											X					
SRATS_011												X				
SRATS_012													X			
SRATS_013													X			
SRATS_014														X		
SRATS_015				X												
SRATS_016					X											
SRATS_017						X										
SRATS_018															X	
SRATS_019																X

Figure 4.7 Traceability matrix between SRATS and HLRs

The final activity of the Requirements phase, referred to as *Requirements Specification and*

Validation, concentrates on validating the HLRs to ensure their robustness and readiness for further development phases. Previously, a set of HLRs was drafted using the SRATS, and these HLRs underwent preliminary verification activities. In this phase, however, a peer review is carried out by an independent reviewer who was not involved in the original HLR drafting. This impartial evaluation is crucial to ensuring that all standards have been satisfied, that the requirements are accurate and unambiguous and that there is complete traceability between each HLR and the system requirements, which includes all relevant parts.

If any issues or inconsistencies are discovered during the peer review, the original author logs a defect for rectification, which initiates an iterative process to enhance the requirements. The reviewer follows a rigorous checklist to confirm that each DO-178C objective is met, including the standards' clarity, consistency, and completeness. This systematic approach ensures that the HLRs meet stringent quality and compliance criteria, reducing the likelihood of errors that could spread to later phases of development. Once all of the checklist elements have been satisfactorily completed, the evaluation is formally documented.

Checklist High-Level Requirements

Review ID:	Peer Review (PR) HLR 001
Review Date:	2024-09-29
Reviewer(s):	
Project Name:	CAS_CollisionAvoidance
Data Item(s) under review:	The purpose of this PR is to review the high-level requirements of the above-mentioned program. Data to be reviewed: Refer to the "HighLevelRequirements" group in TBManager with implemented Baseline defined in the field below. Number of HLRs to review: 16
Software Level:	B
Version / Baseline:	Baseline_005_Revew_HLR
Is it verified ?	Yes

Name	Role	Is the Author?
Henrique Amaral Misson	Sw developer	☐
		☐
		☐

Checklist Item	Yes / No	Comments	Correction
High-level requirements comply with system requirements? (A-3.1)	☑ TRUE		
High-level requirements are accurate and consistent? (A-3.2)	☑ TRUE	No. Refer to defects for more details	Corrected during implementation
High-level requirements are compatible with target computer? (A-3.3)	☑ TRUE		
High-level requirements is verifiable? (A-3.4)	☑ TRUE	No. Refer to defects for HLR that was not verifiable.	Corrected during implementation
High-level requirements conforms to standards? (A-3.5)	☑ TRUE		
High-level requirements are traceable to system requirements? (A-3.6)	☑ TRUE		
Algorithms are accurate? (A-3.7)	☑ TRUE	Not Aplicable - No algorithms defined	
Are all derived requirements, if applicable, are justified?	☑ TRUE		

Figure 4.8 Checklist used for peer review of HLRs during the validation

Figure 4.8 illustrates the peer review checklist used during this validation phase. This checklist ensures that all DO-178C objectives related to HLRs are rigorously evaluated, including compliance with system requirements (A3.1), accuracy and consistency (A3.2), and traceability (A3.6). Any identified issues are documented in the "Comments" section and addressed iteratively during the correction process.

Throughout this process, configuration management is strictly maintained in TBManager, where all review activities, changes, baselines, and defects are recorded. This allows for full documentation of the requirements development process, and TBManager can provide a report summarizing the results of the *Requirements Specification and Validation* phase. The

final output of this phase, known as Requirements Data, consists of both validated HLRs and DHLRs, which will serve as the foundational specifications for the design and coding phases.

The requirements phase concluded with the generation of validated Requirements Data, consisting of SRATS, HLRs, and DHLRs. These outputs formed a robust foundation for the design and coding phases, fulfilling all Table A-3 objectives of DO-178C (Figure 4.9). This iterative and tool-supported process mitigated risks early, ensuring that subsequent phases adhered to the highest standards of safety and certification rigour.

Objectives	Artifacts	Assets	...	Objective Name	Objective Standard	Objective S...	Placeholders
Table A-3 1	0	3	...	High-level requirements comply with system requirements	DO-178C - Populated	Fulfilled	100.00%
Table A-3 2	16	3	...	High-level requirements are accurate and consistent	DO-178C - Populated	Fulfilled	100.00%
Table A-3 3	0	1	...	High-level requirements are compatible with target computer	DO-178C - Populated	Fulfilled	100.00%
Table A-3 4	16	3	...	High-level requirements are verifiable	DO-178C - Populated	Fulfilled	100.00%
Table A-3 5	0	1	...	High-level requirements conform to standards	DO-178C - Populated	Fulfilled	100.00%
Table A-3 6	16	2	...	High-level requirements are traceable to system requirements	DO-178C - Populated	Fulfilled	100.00%
Table A-3 7	0	1	...	Algorithms are accurate	DO-178C - Populated	Fulfilled	100.00%

Figure 4.9 DO-178C objectives for verification of outputs of software requirement process in TBmanager

Design Phase

The design phase of the methodology centred around defining the architecture and the LLRs of the CAS to ensure that it adhered to both operational and safety goals, as outlined in the HLRs.

The design phase begins with an in-depth review of the *Requirements Data* generated during the requirements phase. The goal of this analysis is to decompose the HLRs into modular subsystems, each responsible for a critical function of the CAS. Key subsystems identified include traffic detection, tracking, collision evaluation, threat prioritization, and maneuver determination. For each subsystem, essential inputs, outputs, and interactions with other components and external equipment are carefully described and documented. Once these components and their interfaces have been identified, a preliminary architecture model is developed. This model, created in the Architecture Analysis and Design Language (AADL) [84] using the Osate tool [85], encapsulates both internal relationships and system-level views of CAS components. The design adheres to the ARINC653 [86] industry-standard partitioning scheme through an AADL library.

One of the critical subsystems identified during the design phase is *Traffic Detection*. This subsystem is responsible for continuously monitoring the UAV's surroundings. The architec-

ture of this subsystem was modelled in AADL, as shown in Figure 4.10, capturing its inputs (sensor data from onboard radar and external data sources), outputs (a list of detected traffic), and interactions with other CAS components, such as *Traffic Tracking* and *Collision Evaluation*.

The architectural representation (Figure 4.10) encapsulates the core operations of this subsystem, including surveillance volume calculation and traffic detection. The figure highlights traceability to the associated HLRs (HLR_001 - HLR_004), ensuring that the subsystem aligns with the requirements established in the previous phase. For a detailed view of the architectural diagrams of all the identified components of the CAS Architecture refer to Appendix A.

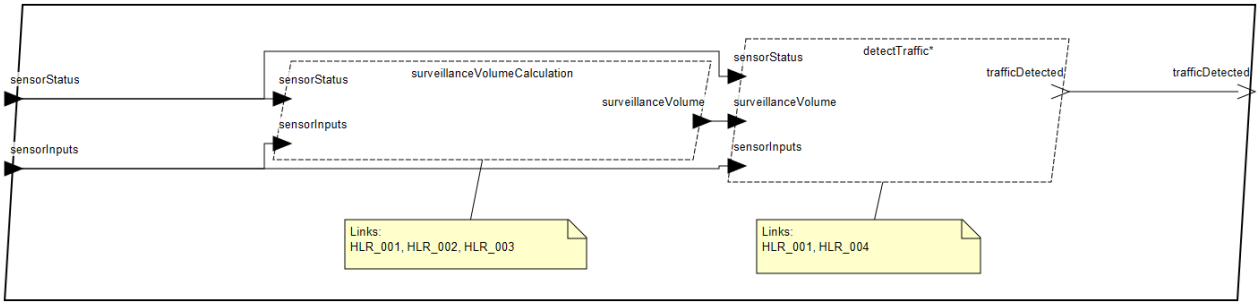

Figure 4.10 AADL model of the traffic detection subsystem

An independent peer review was conducted to confirm that the architecture aligns with the HLRs and meets the DO-178C objectives. This review involved a detailed checklist evaluation to verify the architecture's compliance with key DO-178C objectives, including consistency, verifiability, and traceability to HLRs. Figure 4.11 illustrates the peer review checklist completed during this validation phase. As shown, any defects, such as missing component inputs, were identified and corrected during this process.

In addition, formal verification was performed using model-checking approaches to ensure the architecture's accuracy and reliability. The architecture was modelled in the Promela language and analyzed using the Spin model checker, which converts features of interest into LTL statements. These LTL attributes specify important system behaviours and limitations, guaranteeing that the architecture design meets the necessary safety and functional requirements. The verification process focused on critical system properties, such as liveness properties, ensuring that critical actions (such as avoidance maneuvers) are eventually executed when necessary and safety properties, preventing the activation of incorrect or dangerous maneuvers, thereby avoiding unintended consequences.

Key properties analyzed through LTL claims included:

Checklist Architecture

Review ID:	Peer Review (PR) - Arch 001
Review Date:	
Reviewer(s):	Name: Henrique Amaral Misson; Role: SW developer; Is the Author?
Project Name:	CAS_CollisionAvoidance
Data Item(s) under review:	The purpose of this PR is to review the architecture of the above-mentioned program. Data to be reviewed: Architecture diagrams Number of HLRs to review: 6
Software Level:	2
Version / Baseline:	Baseline_007_LLR
Is it verified ?	Yes

Checklist Item	Yes / No	Comments	Correction
Software Architecture is compatible with high-level requirements? (A-4.8)	TRUE	No - Refer to defects - Missing some component inputs	Missing component inputs were added to all diagrams. Refer to the document for more details.
Software Architecture is consistent? (A-4.9)	TRUE	No - Refer to defects - Missing some component inputs	Missing component inputs were added to all diagrams. Refer to the document for more details.
Software Architecture is compatible with target computer? (A-4.10)	TRUE		
Software Architecture is verifiable? (A-4.11)	TRUE		
Software Architecture conforms to standards? (A-4.12)	TRUE		
Are design decisions clearly documented?	TRUE		
Is the architecture robust?	TRUE		
Software partitioning integrity is confirmed? (A-4.13)	TRUE	Only 1 partition	

Figure 4.11 Checklist used for peer review of the architecture

- c1: A maneuver command is eventually executed for detected collision threats.
  $\Box\Diamond CommandManeuver@SendCmdTrue$

- c2: Detection of traffic leads to a maneuver command.
  $\Box(DetectTraffic@SendTrafficDetected) \rightarrow \Diamond(CommandManeuver@SendCmdTrue)$

- c3: Collision evaluation follows traffic detection.
  $\Box(DetectTraffic@SendTrafficDetected) \rightarrow$
  $\Diamond(CollisionEvaluation@EvaluateCollisionPotential)$

The results showed that no assertion violations or invalid end states were detected, demonstrating that the model satisfies all safety and functional properties in the explored scenarios. However, some claims exhibited unreached states. The analysis of these results indicates that while the core architecture is robust, the unreached states suggest potential areas for refinement or the need for additional test scenarios. For example, claim *c1* revealed missed conditions where detected threats did not lead to a maneuver command, suggesting the need for enhanced conditions in the model. Claim *c3* highlighted edge cases where collision evaluation did not consistently follow traffic detection, pointing to potential gaps in the prioritization logic. These findings were documented and addressed iteratively, ensuring continuous improvement of the architecture model and alignment with the safety-critical objectives of DO-178C.

While architectural design activities occur outside the LDRA tool suite, the verified architecture model and supporting documentation were imported into TBManager. This integration ensured that all verification results were tracked and included in the overall configuration management system, facilitating traceability and certification artifact generation.

As part of the iterative refinement process, the LLR development activities focused on translating the HLRs and the system architecture into detailed, comprehensive specifications that would guide the coding phase. These LLRs were precisely linked to planned software implementations, enabling developers to write functions with exact behaviour and data flow specifications. Each LLR ensured traceability to the HLRs and architecture while meeting DO-178C compliance objectives.

The LLRs were derived to define the operational behaviour of each subsystem with precision. For instance:

- LLR-001 specifies that when the sensor status is active, the TrafficDetection function retrieves the sensor inputs, initiating the data acquisition process.
- LLR-002 establishes the rules for validating sensor inputs, ensuring parameters such as DetectionRange, AzimuthFOR, and ElevationFOR fall within safe operational ranges.

A comprehensive list of the LLRs is provided in Appendix A, which includes LLR-001 through LLR-037.

Formal verification was conducted to validate the accuracy, consistency, and compliance of the LLRs with the HLRs. This process, conducted in two stages, ensured adherence to DO-178C objectives A4.1 (compliance), A4.2 (accuracy), and A4.4 (verifiability).

1. Execution of the Alloy Model Representing All LLRs
   A comprehensive Alloy model was created to represent all LLRs as a cohesive system. This model encoded the functional logic, interactions, and constraints described in the LLRs. Using the Alloy Analyzer, the model was executed to validate internal consistency and interaction across subsystems. The analysis confirmed that the LLRs collectively formed a logically sound design. Instances were generated successfully without contradictions, demonstrating that the specified constraints did not conflict and that subsystems interacted as expected under the architectural model.

2. Assertion-Based Compliance Checks with HLRs
   Once the internal consistency of the LLRs was validated, assertions were defined for each HLR to ensure compliance. These assertions encapsulated key behaviours derived from the HLRs and were checked against the Alloy model. For example:

   - Assertion HLR_001_TrafficDetection verified that the inputs conformed to the correct field of regard (AzimuthFOR and ElevationFOR) for detecting traffic. It is formally expressed as:

```
assert HLR_001_TrafficDetection {
    all td: TrafficDetection |
        some td.sensorInput =>
        td.sensorInput.DetectionRange > 0 and
        td.sensorInput.AzimuthFOR >= -110 and td.sensorInput
           .AzimuthFOR <= 110 and
        td.sensorInput.ElevationFOR >= -15 and td.
           sensorInput.ElevationFOR <= 15
    }
    check HLR_001_TrafficDetection for 5
```

Listing 4.3 HLR_001_TrafficDetection Assertion

The Alloy Analyzer executed this assertion, and no counterexamples were found (as shown in Figure 4.12), confirming compliance with HLR_001. The absence of counterexamples demonstrated that the system adhered to operational parameters for the sensor input's field of regard, ensuring comprehensive spatial coverage and reliable traffic detection.

- Assertion HLR_008_ManeuverDetermination ensured that appropriate maneuvers were determined in response to detected collision threats. The validation of this assertion highlighted the system's ability to proactively determine maneuvers upon detecting potential collisions.

The results of these checks confirmed that all assertions were validated successfully, with no counterexamples identified. The LLRs adhered to their corresponding HLRs, ensuring alignment with operational and safety goals. These results validated the robustness of the LLRs and their compliance with DO-178C objectives, minimizing risks of errors propagating into the coding phase.

```
Executing "Check HLR_001_TrafficDetection for 5"
  Solver=sat4j Bitwidth=4 MaxSeq=5 SkolemDepth=1 Symmetry=20 Mode=batch
  142476 vars. 2948 primary vars. 275239 clauses. 4130ms.
  No counterexample found. Assertion may be valid. 8ms.
```

Figure 4.12 Results for checking the HLR_001_TrafficDetection assertion

Throughout the design phase, automated traceability tools (TBmanager) were employed to ensure that all LLRs could be traced back to their respective HLRs. This traceability

guaranteed that any changes made during the design process were reflected throughout the system, maintaining alignment with DO-178C objective A4.6 (traceability). TBmanager was used to generate and maintain traceability matrices, linking HLRs to their corresponding LLRs. These matrices ensured that each requirement was traceable from its specification to its detailed design. As refinements were made during verification activities, the matrices were continuously updated to reflect the most current relationships between HLRs and LLRs. The integration of automated tools allowed for real-time compliance checks against DO-178C requirements. This approach ensured that the evolving design consistently adhered to certification standards, minimizing the risk of misalignment between HLRs and LLRs during iterative updates.

Figure 4.13 illustrates the traceability matrix generated, which links HLRs to their associated LLRs. The green markers indicate relationships between requirements, ensuring complete coverage and traceability as mandated by DO-178C standards.

Requireme...	LLR-001, Sen...	LLR-002, Sen...	LLR-003, Get...	LLR-004, Get...	LLR-005, Get...	LLR-006, Extr...	LLR-007, Con...	LLR-008, Veri...	LLR-009, Out...	LLR-013, Det...	LLR-014, Est...	LLR-015, Upd...	LLR-016, Mai...	LLR-017, Trac...	LLR-018, Coll...	LLR-019, Thr...	LLR-020, Tim...
HLR_001	X	X	X	X	X	X	X	X	X								
HLR_002		X															
HLR_003		X															
HLR_004										X							
HLR_005											X	X	X	X			
HLR_006															X	X	
HLR_007																	X
HLR_008																	
HLR_009																	
HLR_010																	
HLR_011																	
HLR_012																	
HLR_013																	
HLR_014																	

Figure 4.13 Traceability matrix between HLRs and LLRs

The FMEA and FTA activities were crucial in identifying potential failure points and assessing their impact on the CAS. These analyses strengthened the robustness and reliability of the system, uncovering failure modes that could compromise safety and functionality.

The FMEA focused on identifying potential failure modes across each subsystem of the CAS, evaluating their causes, and analyzing their effects. This systematic approach prioritized failure modes based on their severity and proposed mitigations to address vulnerabilities. For example, in the *Traffic Detection subsystem*, failure modes such as *"Traffic detected late"* or *"Traffic detected at incorrect height"* were analyzed. These failures were assessed for their impact on the UAV's ability to respond to collision threats in time. Recommendations included implementing dual-sensing technologies, periodic sensor calibration, and improving data validation protocols (see Appendix A for the complete FMEA report).

Complementing the FMEA, the FTA provided a graphical representation of fault scenarios

and their contributing factors. By tracing failures back to their root causes, the FTA highlighted dependencies and vulnerabilities in the system architecture. For the *Traffic Detection* subsystem, the FTA (Figure 4.14) identified key contributors to detection failures, including sensor failure, software errors, communication failures, configuration errors, and physical obstruction.

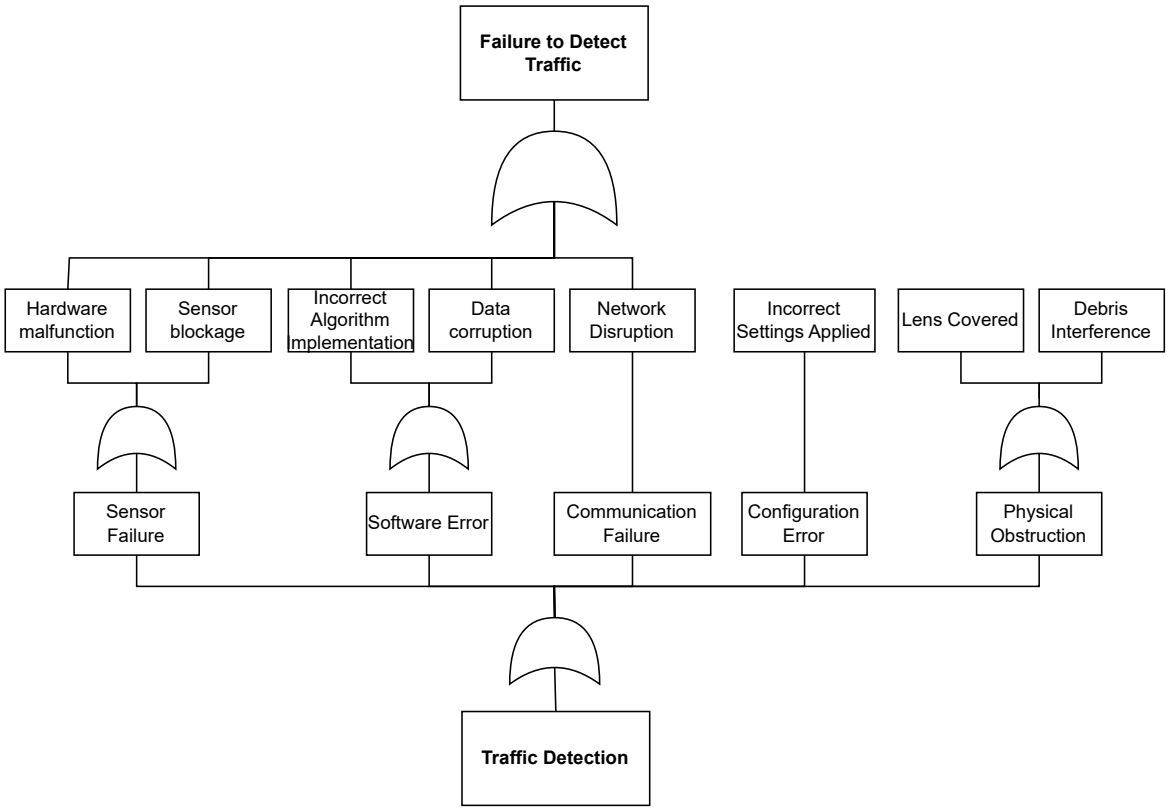

Figure 4.14 FTA for the Traffic Detection subsystem

The final activity in the design phase involved a comprehensive review of the CAS architecture and design documentation to ensure conformance to all safety and quality standards. This step aimed to validate the readiness of the system design for the subsequent coding phase, ensuring that all critical safety and operational requirements were met.

To verify the completeness, accuracy, and consistency of the design documentation, independent peer reviews were conducted. These reviews focused on validating that:

- The design aligned with the safety-critical objectives outlined in DO-178C.
- All design artifacts were traceable to their respective HLRs and LLRs.

- The architecture and LLRs were robust and sufficiently detailed to guide the coding phase.

Like the HLR and architecture reviews, the peer reviews employed detailed checklists to ensure systematic evaluation.

By the conclusion of this phase, all defects identified during the reviews and verification activities were resolved. Figure 4.15 presents a summary of the DO-178C objectives fulfilled during the design phase. This table, generated using TBManager, demonstrates 100% compliance with objectives related to LLRs and software architecture, including traceability, consistency, and verifiability. The robust alignment between design activities and certification requirements further validated the readiness of the design for the coding phase.

Objectives	Artifacts	Assets	...	Objective Name	Objective Standard	Objective S...	Placeholders
Table A-4 1	0	3	...	Low-Level requirements comply with high-level requirements	DO-178C - Populated	Fulfilled	100.00%
Table A-4 2	34	2	...	Low-level requirements are accurate and consistent	DO-178C - Populated	Fulfilled	100.00%
Table A-4 3	0	1	...	Low-level requirements are compatible with target computer	DO-178C - Populated	Fulfilled	100.00%
Table A-4 4	34	2	...	Low-Level requirements are verifiable	DO-178C - Populated	Fulfilled	100.00%
Table A-4 5	0	1	...	Low-level requirements conform to standards	DO-178C - Populated	Fulfilled	100.00%
Table A-4 6	50	2	...	Low-level requirements are traceable to high-level requirements	DO-178C - Populated	Fulfilled	100.00%
Table A-4 7	0	1	...	Algorithms are accurate	DO-178C - Populated	Fulfilled	100.00%
Table A-4 8	0	3	...	Software architecture is compatible with high-level requirements	DO-178C - Populated	Fulfilled	100.00%
Table A-4 9	0	3	...	Software architecture is consistent	DO-178C - Populated	Fulfilled	100.00%
Table A-4 10	0	1	...	Software architecture is compatible with target computer	DO-178C - Populated	Fulfilled	100.00%
Table A-4 11	0	3	...	Software architecture is verifiable	DO-178C - Populated	Fulfilled	100.00%
Table A-4 12	0	2	...	Software architecture conforms to standards	DO-178C - Populated	Fulfilled	100.00%
Table A-4 13	0	2	...	Software partitioning integrity is confirmed	DO-178C - Populated	Fulfilled	100.00%

Figure 4.15 DO-178C objectives for verification of outputs of the software design process in TBmanager

By the end of the design phase, the CAS architecture and LLRs have been thoroughly verified and validated, ensuring that they meet all necessary safety and operational requirements. This provided a robust foundation for the subsequent coding phase, where the verified design would be translated into executable code.

Coding Phase

The Coding phase of the software development process focuses on translating the system design into executable source code. Given the critical nature of the software, a hybrid method was used, combining automated code creation with manual development.

The CAS's functional behaviour was initially modelled in MATLAB Simulink [87], following

the specifications outlined in LLRs, DLLRs, and the overall architecture. Simulink models provided a detailed representation of each subsystem, encapsulating their individual behaviours, data flows, and interactions. Figure 4.16 illustrates the Simulink model of the CAS, showcasing its modular subsystems and data flows. It highlights how functional components were structured and interlinked to achieve the overall operational objectives of the system.

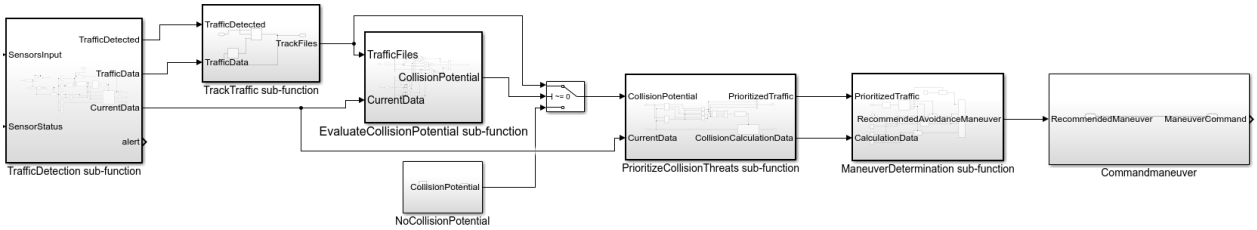

Figure 4.16 Simulink model of the CAS showing subsystems and data flow.

Within Simulink, tests were run to identify and eliminate any typos, logical errors, or inconsistencies in the model, creating a solid basis for code creation. Once validated, the model was translated to C code with Embedded Coder, which created separate .c and .h files for each subsystem, resulting in modular code that is consistent with the system's architectural design.

Although automated code generation formed the majority of the implementation, manual coding was required to integrate the subsystems into a unified software application. This involved programming the interactions and dependencies between subsystems, such as data interchange, control flow, and module timing. These adjustments supplement the automated programming and provide more flexibility in managing complicated interactions inside the system. The generated code complies with DO-178C standards for safety-critical systems, giving close attention to traceability, modularization, and design descriptions.

To preserve traceability, all manual code adjustments were rigorously documented, linking each modification to its corresponding requirements in the LLRs and architectural design. This comprehensive documentation ensured continued adherence to DO-178C objectives, particularly A5.5 (traceability of source code to design).

To integrate the generated and manually adjusted code into the project's configuration and verification framework, all code files were imported into TBManager. This integration enabled comprehensive traceability by mapping code functions back to the LLRs and, subsequently, to the original SRATS through HLRs. As depicted in Figure 4.17, TBManager provides a relational view illustrating the connections between the three levels of requirements and the associated source code procedures.

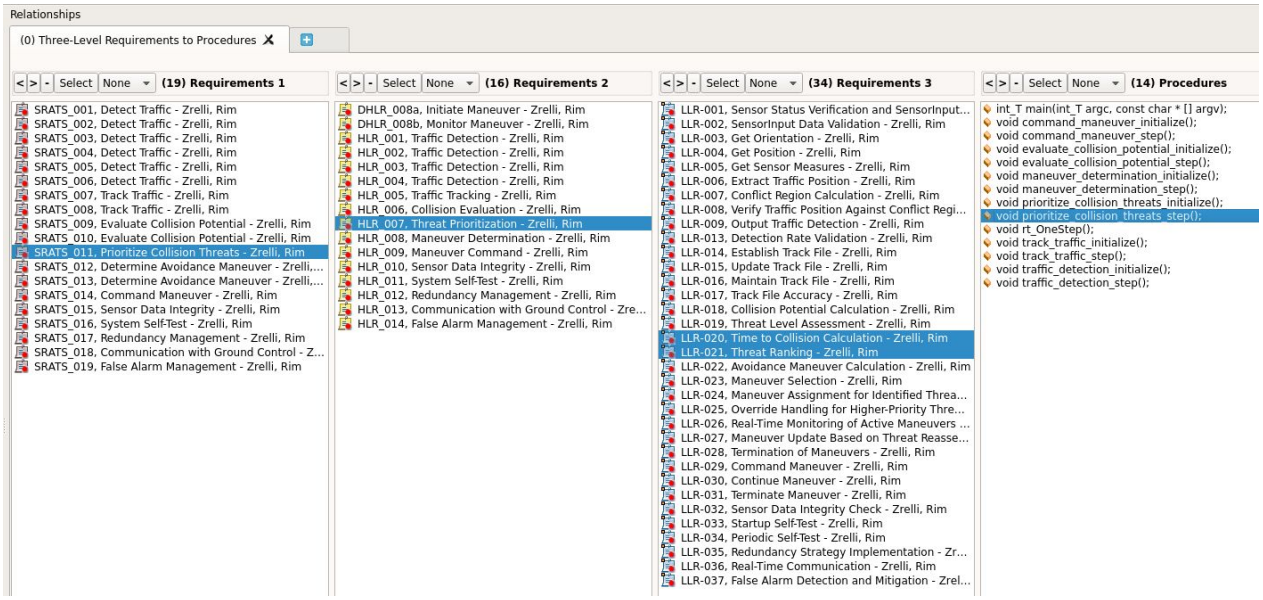

Figure 4.17 TBmanager relationship view showing the traceability between three-requirement levels (SRATS, HLR and LLR) and the source code functions

This traceability structure establishes a clear lineage from high-level system specifications to the implemented code, ensuring that each code function is precisely aligned with the design. It allows for extensive verification, simplifies maintenance tasks, and ensures compliance with DO-178C objectives, particularly A5.5 (traceability of source code to requirements). By maintaining this level of traceability, the project guarantees that the implementation remains consistent with the original design and functional objectives while supporting efficient identification and resolution of potential issues during future updates or audits.

To ensure the correctness and reliability of the code, both static and dynamic code analyses were conducted using the LDRA tool suite.

The generated and manually refined code underwent static analysis to detect potential coding standard violations, evaluate code quality, identify data flow issues, and detect runtime anomalies. LDRA's TBvision tool was employed to conduct this analysis, generating comprehensive static analysis reports that flagged potential violations and ensured adherence to DO-178C objectives A5.4 (conformity to standards) and A5.6 (accuracy and consistency). The analysis began by verifying the source code for conformance with MISRA C 2012 [88], a widely used coding standard for embedded systems. The choice is made as Embedded Coder that generated source code uses this standard.

The static analysis began with an initial scan where TBvision found multiple breaches classified as "required", which must be addressed to comply with the standard. This initial as-

sessment marked the start of an iterative procedure in which violations were systematically addressed and then reanalyzed to assure compliance. During these cycles, the interactive refinement of the code resolved the required violations, allowing for increasing conformity with the standard. In the final revisions, all the violations were treated. Achieving coding standard compliance ensures the source code adheres to a predetermined set of guidelines. Part of the results of the code review are summarized in Figure 4.18. Procedures marked as "PASS" indicate that the referred procedures fully adhered to the standard.

Procedure Results +/-

Code Review Result	Procedure	Source File	Unique Violations	Failure Density (Viols/R.Line %)
PASS	Global Program			
PASS	command_maneuver_step	command_maneuver.c	0%	0%
PASS	command_maneuver_initialize	command_maneuver.c	0%	0%
PASS	evaluate_collision_potential_step	evaluate_collision_potential.c	0%	0%
PASS	evaluate_collision_potential_initialize	evaluate_collision_potential.c	0%	0%
PASS	rt_OneStep	main.c	0%	0%

Figure 4.18 Code review report generated by LDRA

Once compliance with coding standards was achieved, the focus shifted to assessing software quality through a structured evaluation of key metrics: clarity, maintainability, and testability.

- **Clarity**: Ensures the code is easily understandable, facilitating peer reviews, debugging, and future modifications.
- **Maintainability**: Evaluates the ease of implementing changes in response to evolving requirements, ensuring adaptability and resilience over the software lifecycle.
- **Testability**: Measures how readily the code can be tested, which is crucial for ensuring functional accuracy and identifying defects efficiently.

Using LDRA's analysis tools, each metric was thoroughly evaluated, and the results are depicted in Figure 4.19. The analysis revealed that most source files scored exceptionally well in maintainability and testability, indicating a robust design. While the clarity metric showed areas for improvement in specific files, these did not impact compliance or functionality, but they highlight opportunities for further refinement to enhance code readability.

Dynamic Code Analysis: The dynamic analysis of the code concentrates on determining structural coverage, which evaluates the extent to which software code is exercised under certain test conditions. This analysis provides a dynamic metric essential for ensuring the

File Results

File	All Metrics	Clarity	Maintainability	Testability
command_maneuver.c	91%	71%	100%	100%
evaluate_collision_potential.c	91%	71%	100%	100%
main.c	91%	79%	100%	93%
maneuver_determination.c	100%	100%	100%	100%
prioritize_collision_threats.c	96%	86%	100%	100%
track_traffic.c	96%	86%	100%	100%
traffic_detection.c	96%	93%	91%	93%

Figure 4.19 Code quality report generated by LDRA

thoroughness of testing. These criteria are matched to the criticality of the code under test, ensuring proportionate efforts to achieve acceptable coverage. Automated tools, such as LDRA, are strongly suggested for this process due to their efficiency and accuracy, particularly in complex systems.

The structural coverage analysis was carried out using LDRA's TBvision tool. The process entailed instrumenting the source code, in which LDRA added additional code to monitor execution during testing. We provided simulated inputs to the application execution representing UAV position data and sensor detections, modelling various scenarios of traffic presence or absence. These inputs tested various paths and circumstances within the code. The analysis produced extensive results, as shown in Figure 4.20, that included percentage coverage for key metrics including Statement and Branch/Decision coverage for each file, as well as specific information about untested code paths.

The evaluation was conducted to meet the requirements of DO-178C Level B software, confirming that the coverage metrics were appropriate for this criticality level. The assessment confirmed if the required structural coverage parameters for this safety level were met. While the procedure needed careful planning and iterative testing to resolve coverage gaps, LDRA's thorough insights helped to improve code quality and ensure compliance with stringent avionics software standards. This emphasizes the role of dynamic analysis in showing software dependability and functional safety in critical systems.

The code coverage results provide useful information about the UAV software system's performance and reliability. While 100% Statement and Branch/Decision coverage across all files is desirable, the existing results identify areas for improvement and serve as a benchmark. For example, in *maneuver_determination.c*, which obtained 62% Statement and 39% Branch/Decision coverage, the analysis highlighted particular regions where Simulink-generated support functions like *rtIsNaNF* were not exercised during testing. These findings demonstrate the LDRA analysis's effectiveness in identifying potential optimization opportunities, even inside automatically generated code.

System Summary

Name	Statement (%)	Branch/Decision (%)
CollisionAvoidance	88%	65%

File Summary

Name	Statement (%)	Branch/Decision (%)
command_maneuver.c	100%	100%
evaluate_collision_potential.c	100%	70%
main.c	100%	77%
maneuver_determination.c	62%	39%
prioritize_collision_threats.c	100%	88%
track_traffic.c	100%	100%
traffic_detection.c	100%	69%

Figure 4.20 Code coverage analysis report generated by LDRA

The initial results of the code coverage analysis performed using TBvision, with a set of input data, revealed lower-than-expected Statement and Branch/Decision coverage percentages for several files. However, LDRA allows us to exercise the code using test cases run in TBrun, allowing us to refine and expand our testing. We estimate that by using TBrun to run more or more specialized test cases, we will improve the coverage results for the bulk of the files, resulting in more extensive code validation.

Overall, the results demonstrate the effectiveness of using automated techniques in UAV software development. The coverage attained shows that essential areas of the code were thoroughly tested, with detailed reports providing actionable insights for targeted improvements. Importantly, the procedure validates the idea of employing LDRA to improve the rigor and efficiency of the development process.

Formal verification was employed during the coding phase to ensure that the CAS code adhered to critical safety properties, aligning with DO-178C objectives for verifiability (A5.3) and accuracy (A5.6). Bounded Model Checking [89] was utilized to validate the correctness of code segments by modelling the code logic and verifying it against pre-defined safety properties. This approach provided mathematical assurance of the system's logical soundness and operational reliability.

To this intention, the Efficient SMT-Based Bounded Model Checker (ESBMC) was chosen for its compatibility with the C programming language and its capability to simulate program execution across all possible input scenarios. ESBMC is designed to detect a wide array of potential programming issues, including out-of-bounds array accesses, null or invalid pointer

dereferences, memory alignment violations, integer overflows, and floating-point anomalies (e.g., NaNs and division by zero). Additionally, it aids in identifying memory leaks, a critical factor in ensuring code robustness and compliance with DO-178C standards for resilience and dependability.

To use ESBMC for verifying the correctness of our code, the first step is to specify input handling with nondeterministic values that ESBMC can explore during bounded model checking. Specifically, sensor values inputs were defined as *nondet_double()*, ensuring that the model checker can test a wide range of potential values. Furthermore, $___ESBMC_assume$ is used to apply constraints on the sensor inputs, limiting the values to a suitable range depending on the system's predicted behaviour. This is important because ESBMC works by investigating every possible combination of inputs, and it needs deterministic inputs to verify every possible program execution path.

The verification process leveraged ESBMC's ability to analyze and prove that all states within the program were reachable under the forward condition. This comprehensive analysis was exemplified by the successful verification output shown in Figure 4.21, which demonstrates that the CAS code satisfied all specified properties after exhaustive exploration of execution paths, with a maximum bounded state depth of $k = 33$.

```
No solver specified; defaulting to Boolector
Encoding remaining VCC(s) using bit-vector/floating-point arithmetic
Encoding to solver time: 0.084s
Solving with solver Boolector 3.2.0
Encoding to solver time: 0.084s
Runtime decision procedure: 0.000s
BMC program time: 0.520s

VERIFICATION SUCCESSFUL

Solution found by the forward condition; all states are reachable (k = 33)
```

Figure 4.21 Bounded Model Checking results using ESBMC for CAS code

The final step in the coding phase involved a thorough review and automated testing to ensure that the code met all safety and performance standards. LDRA was employed to conduct automated testing and generate test reports, ensuring the code's robustness and compliance.

- Code Review: The generated and modified code was systematically reviewed by experts to ensure that it adhered to the system's functional and safety requirements. The review process focused on verifying that the code conformed to the architectural models and complied with DO-178C objective A5.1 (compliance with LLRs).

- Automated Testing: Automated test cases were performed using TBrun, where a sequence of test cases was designed and executed to test the source code thoroughly. TBrun enables the execution of procedure calls using the specified test data, guaranteeing that the source code works as intended. The test data was defined using test cases drawn from HLRs and LLRs, ensuring complete coverage of the system's requirements. In TBrun, a white-box analysis was performed, with an emphasis on a thorough structural evaluation of the code under test. This analysis used Coverage Analysis to determine the usefulness of the test data in running the unit tests and guaranteeing the code's functionality.

  A total of 44 test cases were run, with the results indicating that 100% of them passed properly, as depicted in Figure 4.22. A manual review was also performed to confirm that the expected outputs were reached, adding another layer of validation to the tests' accuracy and completeness. This comprehensive method not only evaluated the software's conformance to functional requirements but also helped to ensure the UAV system's overall reliability and safety. These tests ensured that each component functioned correctly and that the code complied with DO-178C objectives A5.1 (compliance with LLRs) and A5.2 (compliance with software architecture).

TBrun Unit / Module Test

Name of Sequence	Test Cases	Box Mode	Regression Analysis
CAS_TCI_sequence	44	White	100% 44 Pass

Figure 4.22 Tbrun Unit / Module Test report

By the end of the coding phase, the CAS source code was fully generated, verified, and validated through a combination of formal methods, manual reviews, and automated testing. Figure 4.23 provides a summary of how the code satisfied DO-178C coding objectives. This rigorous process ensured the code adhered to all required safety and performance standards, forming a robust and compliant foundation for the subsequent Software Integration phase.

4.4.3 Data Collection and Analysis

This subsection synthesizes the verification and validation (V&V) findings to highlight the CAS's compliance with DO-178C objectives. It consolidates evidence supporting the accuracy, safety, and robustness of the development lifecycle, emphasizing key metrics, traceability, and certification readiness.

Objectives	Artifacts	Assets	...	Objective Name	Objective Standard	Objective S...	Placeholders
Table A-5 1	42	2	...	Source Code complies with low-level requirements	DO-178C - Populated	Fulfilled	100.00%
Table A-5 2	8	2	...	Source Code complies with software architecture	DO-178C - Populated	Fulfilled	100.00%
Table A-5 3	8	2	...	Source Code is verifiable	DO-178C - Populated	Fulfilled	100.00%
Table A-5 4	3	3	...	Source Code conforms to standards	DO-178C - Populated	Fulfilled	100.00%
Table A-5 5	42	1	...	Source Code is traceable to low-level requirements	DO-178C - Populated	Fulfilled	100.00%
Table A-5 6	8	2	...	Source Code is accurate and consistent	DO-178C - Populated	Fulfilled	100.00%
Table A-5 7	0	1	...	Output of software integration process is complete and correct	DO-178C - Populated	Fulfilled	100.00%
Table A-5 8	0	2	...	Parameter Data Item File is correct and complete	DO-178C - Populated	Fulfilled	100.00%
Table A-5 9	0	1	...	Verification of Parameter Data Item File is achieved	DO-178C - Populated	Fulfilled	100.00%

Figure 4.23 DO-178C objectives for verification of outputs of the software coding process in TBmanager

Verification and Validation Findings

The V&V process, conducted iteratively across development phases, employed formal methods, peer reviews, and automated tools, yielding high-confidence results. The following points summarize critical outcomes:

1. **Comprehensive system verification**:
   - Formal verification of SRATS, HLRs, and LLRs using the Alloy Analyzer validated logical soundness and consistency. Assertions such as *CompleteTrafficDetection* ensured the CAS accurately detected and tracked traffic under defined constraints (e.g., azimuth and elevation fields of regard).
   - Architectural verification through Spin confirmed compliance with key safety properties, including liveness (ensuring maneuvers are eventually executed for collision threats) and safety (preventing unsafe maneuvers).
2. **Code integrity**:
   - Static code analysis using LDRA TBvision highlighted initial mandatory MISRA C 2012 violations, resolved iteratively to achieve near-complete compliance with only minor advisory warnings. These results ensured the source code aligned with DO-178C objectives for accuracy, consistency, and adherence to coding standards.
   - Formal methods applied to the executable code (via ESBMC) verified memory safety and correctness, identifying and resolving potential anomalies such as null pointer dereferences and integer overflows. These checks fortified the code's resilience against runtime faults.

3. **Dynamic Testing**:

   - Tbvision and TBrun were used to perform dynamic code analysis for coverage, where particular data inputs were supplied to assess the code's execution. The structural (or code) coverage, which evaluates the sections of the code that were exercised during testing, was the main focus of this investigation. The findings showed significant coverage, with most of the code functions being successfully executed with the given inputs. To guarantee that all functionalities are fully tested and to attain more extensive code coverage, the analysis also pointed out areas that require improvement.

     Alongside those analyses, test cases were created and run in accordance with the system's requirements, guaranteeing that every test matched particular HLRs and LLRs. By matching the test cases to these specifications, the testing procedure proved that the system satisfies its stated goals by validating the code's functionality and demonstrating thorough requirements coverage.

Traceability and Compliance Evidence

Traceability underpinned the CAS development, ensuring all requirements remained aligned through automated tools. Evidence includes:

1. **Traceability Matrices**:

   - TBManager generated bidirectional traceability matrices linking SRATS to HLRs (Figure 4.7), HLRs to LLRs (Figure 4.13), and LLRs to source code (Figure 4.24). The matrices demonstrated full coverage and allowed efficient impact analysis for requirement changes.

2. **Coverage Reports**:

   - The LDRA *Project Coverage Summary* (Figure 4.25) provides a comprehensive overview of the coverage achieved across all levels of requirements (SRATS, HLRs, and LLRs) and the associated test cases. The report highlights the relationships and traceability between requirements and test cases, with each set organized into specific groups. The report confirmed full traceability from SRATS to HLRs and LLRs, ensuring that every software requirement was adequately addressed and tested during the verification phase. The report shows a Project Coverage of 100%, indicating that all requirements and test cases across the Groups have

been covered. Achieving this level of coverage validates the completeness of the verification activities and ensures compliance with certification standards.

3. **Defect Management**:

   - Automated defect reporting tracked issues across phases, with key findings recorded in a consolidated *Defect Summary Report*. Figure 4.26 illustrates resolved defects, categorized by severity, confirming adherence to iterative improvement principles.

Requireme...	int_T main(in...	void comma...	void comma...	void evaluat...	void evaluat...	void maneuv...	void maneuv...	void prioritiz...	void prioritiz...	void rt_OneS...	void track_tr...	void track_tr...	void traffic_d...	void traffic_d...
LLR-001													X	X
LLR-002													X	X
LLR-003													X	X
LLR-004													X	X
LLR-005													X	X
LLR-006													X	X
LLR-007													X	X
LLR-008													X	X
LLR-009													X	X
LLR-013													X	X
LLR-014											X	X		
LLR-015											X	X		
LLR-016											X	X		
LLR-017											X	X		
LLR-018				X	X									
LLR-019				X	X									
LLR-020								X	X					
LLR-021								X	X					
LLR-022						X	X							
LLR-023						X	X							
LLR-024						X	X							
LLR-025						X	X							
LLR-026						X	X							
LLR-027						X	X							
LLR-028						X	X							
LLR-029		X	X											

Figure 4.24 Traceability matrix between LLRs and Code

LDRA TBmanager Project Coverage Summary

Project	/home/formal/ldra_server/DO_development/Documents/CollisionAvoidance_Project_LDRA/UAV_CollisionAvoidance.tbp	Date	Thu 28 Nov 2024 01:42:25 PM	Version	10.0.3

Project Summary +/-

Project Coverage	100%		
Total Number of Requirement/Test Cases	140	Total Number of Uncovered Requirements	0
Total Number of Coverable Requirements	69	Total Number of Covered Requirements	69
Number of Group(s)	5	Number of Coverable Group(s)	3

Figure 4.25 LDRA TBmanager project coverage summary for CAS requirements and test cases

Defects

Parents	Defect Report Number	Priority	DR Description	Status
(2) HLR_008, HLR_009	PR_HLR_Defect_001	Medium	Defects found during Alloy Specifi...	Closed
(12) HLR_001, HLR_002, HLR_003, HLR...	PR_HLR_Defect_002	Low	A peer-review was done and Alloy...	Closed
(2) DHLR_008a, DHLR_008b	PR_HLR_Defect_003	Low	A peer-review was done and Alloy...	Closed
(1) LLR-002	PR_LLR_Defect_001	Medium	Traceability? This LLR traces HLR_...	Closed
(2) LLR-013, LLR-016	PR_LLR_Defect_003	Medium	Those LLR are too high-level. The...	Closed
(1) LLR-014	PR_LLR_Defect_004	Medium	Why "establish" meas? Try to use ...	Closed
(1) LLR-017	PR_LLR_Defect_005	Medium	This LLR is too high level. - The u...	Closed
(2) LLR-025, LLR-027	PR_LLR_Defect_006	Medium	Defect found during Alloy Specific...	Closed
(26) LLR-001, LLR-003, LLR-004, LLR-0...	PR_LLR_Defects_007	Low	A peer-review was done and an Al...	Closed
(2) SRATS_012, SRATS_013	PR_SASR_Defect_001	Medium	Defects found during Alloy Specifi...	Closed
(17) SRATS_001, SRATS_010, SRATS_01...	PR_SASR_Defects	Low	A peer-review was done and an Al...	Closed
(1) LLR-015	PR_LLR_Defect_008	Medium	What is the X updates per second...	Closed

Figure 4.26 Defect summary report

The aggregated results underscore the readiness of the CAS for DO-178C certification, supported by a comprehensive suite of evidence. Verified traceability artifacts clearly demonstrate alignment with DO-178C objectives, ensuring that requirements, design elements, and implementation remain consistently connected throughout the development lifecycle. Formal verification results further validate the logical integrity and correctness of the system, addressing critical safety and functional properties. Additionally, extensive static and dynamic analysis outputs affirm the runtime reliability of the system, confirming its conformance to industry standards and enhancing confidence in its operational robustness.

4.5 Results and Discussion

4.5.1 Evaluation Against DO-178C Objectives

The methodology's effectiveness in achieving compliance with DO-178C was evaluated by systematically addressing its objectives through rigorous verification and validation activities. The process integrated formal methods with automated tools, ensuring that development phases, from requirements to coding, adhered to certification standards. Table 4.3 provides a consolidated view of the verification activities, tools employed, outcomes generated, and corresponding DO-178C objectives.

Key milestones in this evaluation include:

1. **Verification of Requirements and traceability**: The development began with the formal verification of SRATS, which were refined to eliminate inconsistencies and ambiguities. Tools such as Alloy ensured that HLRs and detailed LLRs maintained logical integrity and precise alignment with system objectives. Traceability was managed comprehensively across all requirement levels using TBmanager, which facilitated the

generation of traceability matrices. These matrices confirmed bidirectional traceability essential for compliance with DO-178C objectives A3.6 and A4.6.

2. **Design phase evaluation**: The architectural models, developed using AADL and verified with SPIN, were validated for logical consistency and compliance with system constraints. Liveness and safety properties were analyzed through LTL claims to confirm the correctness of subsystem interactions. The iterative use of model-checking tools provided continuous refinement of design artifacts, addressing objectives A4.8, A4.9 and A4.11.

3. **Code verification and static/dynamic analysis**: The coding phase was fortified through automated static and dynamic analysis using TBvision. Static analysis identified and resolved coding standard violations, ensuring conformity with MISRA C 2012, code quality (clarity, maintainability and testability) and alignment with DO-178C objective A5.4. Dynamic testing validated runtime behaviour, addressing code coverage. Formal verification, supported by ESBMC, further ensured the logical consistency and verifiability of the implemented code, as required by A5.3.

4. **Review processes and certification artifacts**: Peer reviews were employed during the requirements, design, and coding phases providing an independent evaluation of artifacts against DO-178C criteria. These reviews, coupled with automated traceability and documentation tools, ensured that certification deliverables such as traceability matrices and accomplishment summaries adhered to DO-178C objectives.

The comprehensive approach applied to the CAS project demonstrates the efficacy of combining formal methods and automation to address DO-178C requirements. By leveraging a structured methodology, the development process achieved early defect detection, robust traceability, and seamless generation of certification artifacts. These outcomes underscore the scalability and effectiveness of the methodology in certifying complex safety-critical systems, paving the way for its application in broader domains.

4.5.2 Impact of Methodology

The proposed methodology represents a significant advancement in addressing the challenges of developing DO-178C-compliant UAV software. By integrating formal methods with automated verification tools, this approach enhances both the rigour and efficiency of the software lifecycle, directly impacting compliance, traceability, and overall system safety.

Enhancements to Certification Compliance

The integration of formal methods into the requirements and design phases ensures early detection of ambiguities and inconsistencies. For example, Alloy models were employed to validate logical soundness in both SRATS and HLRs, while Spin confirmed liveness and safety properties in the software architecture. These formal verification activities streamline adherence to DO-178C objectives such as verifiability, accuracy and consistency. This contrasts sharply with traditional methods that rely heavily on manual reviews, which can miss subtle errors, particularly in complex systems.

The methodology also leverages qualified tools like the LDRA suite to automate and guarantee traceability, structural coverage analysis, and certification evidence generation. This automation addresses the objectives of traceability, verifiability and conformity to standards reducing manual effort and improving the reliability of deliverables. Comparatively, the study on software-defined radio systems [58] highlights the potential for automated testing in certification processes but underscores the limitations of partial tool integration. In contrast, our methodology integrates both formal and automated approaches seamlessly, yielding a more robust certification process.

Efficiency Gains in Safety-Critical Development

The unified framework of formal methods and automated tools enhances efficiency by reducing redundant tasks and manual oversight. For example, TBmanager automates traceability matrix generation, linking requirements to code functions and verification results. Similarly, automated testing through LDRA reduces the time for structural coverage validation and compliance checks. This efficiency is particularly beneficial in meeting the stringent timelines often associated with UAV certification projects, as evidenced in the study [48], which highlighted the resource intensity of certifying adaptive learning agents under DO-178C.

In terms of lifecycle coverage, the use of MBD methodologies complements formal verification by providing a visual and systematic representation of requirements and architecture. The work by [45] also acknowledges the scalability of MBD approaches, especially when integrated with domain-specific modelling languages. However, their findings emphasize the necessity of aligning models with certification requirements, a gap our methodology addresses by embedding formal validation within the MBD process.

Addressing Complex System Challenges

The proposed methodology effectively mitigates the challenges posed by system complexity, especially for UAVs with adaptive or autonomous functionalities. As demonstrated in the case study on the CAS, formal methods ensure the deterministic behaviour of critical functions like maneuver determination. This aligns with recommendations by the study [59], who advocate for model checking and abstraction techniques to ensure compliance with DO-178C in autonomous systems.

Comparative Insights and Limitations

Compared to alternative methodologies, such as the process-oriented build tool for airborne software development discussed by [57], our approach demonstrates greater integration and automation. While the process-oriented tool excels in managing configurations and streamlining testing, it does not incorporate the formal rigour necessary for early-stage error detection. By combining these strengths, our methodology presents a balanced approach to achieving both precision and scalability.

However, the reliance on qualified tools poses a limitation, as the certification process becomes tool-dependent. This highlights a need for further exploration into tool qualification strategies to mitigate dependency risks. Additionally, while formal methods offer unparalleled rigour, their application requires specialized expertise, which can increase the initial learning curve for teams transitioning from traditional practices.

4.5.3 Case Study Insights

The implementation of the CAS for UAVs presented several challenges and required iterative refinements to address discrepancies uncovered during formal verification and automated analyses. This section outlines the key insights gained during the development process, focusing on lessons learned from applying the integrated methodology of formal methods and LDRA tools, challenges encountered, and refinements made to achieve compliance with DO-178C standards.

Challenges Encountered

One of the primary challenges was to manage virtual subsystems faced during code generation using Embedded Coder from the Simulink model. When attempting to generate code for the entire model, which comprised several subsystems, many of these were treated as

virtual subsystems, resulting in no code being generated for them. The approach used to solve this problem was to generate code for every subsystem separately, making sure that all required code was generated. Each subsystem's independently generated code was subsequently manually combined to provide a comprehensive and useful system implementation. While this approach required additional effort in integration, it ensured that all subsystems were properly represented in the final code.

Another significant challenge was related to performing test coverage analysis for the generated code. Initially, there were difficulties in configuring the build environment within LDRA to support the analysis. After this configuration was in place, preliminary results showed that coverage levels were below expectations and that numerous statements and branch/decision points were not covered. Addressing this required analyzing the code in detail and identifying additional possibilities for sensor inputs that could be streamed to improve coverage. The outcomes demonstrated significant improvements once the strategy was put into practice and several iterations were carried out in Tbvision and TBrun. However, some functions generated by Simulink to support the logic—such as utility functions, remained unexercised, which prevented the branch/decision coverage from reaching optimal levels.

Additionally, the initial code analysis using LDRA TBvision revealed numerous coding standard violations and traceability gaps, emphasizing the need for iterative corrections. Early phases involved extensive learning to utilize LDRA's features effectively, including structural coverage analysis and traceability management. The absence of complete familiarity with tool functionalities posed delays, highlighting the importance of training and incremental tool adoption in complex projects.

Iterative Refinements

To resolve uncovered issues, iterative refinements were applied at multiple levels:

1. Refining prioritization logic: Formal verification using Alloy and model-checking tools pinpointed unresolved states in prioritization. Specific refinements included:
    - Clearly defining activation thresholds for collision maneuvers.
    - Introducing dynamic conditions for prioritizing simultaneous threats, ensuring robustness against edge cases.
2. Subsystem integration: The separately generated code for each subsystem was then manually integrated, allowing for a complete and functional implementation of the system.

3. Enhanced test generation: A systematic approach to generating test cases was adopted, leveraging path analysis capabilities in TBvision and TBrun to cover untested branches. Automated testing was integrated with dynamic analysis tools to iteratively close coverage gaps, ultimately achieving compliance with DO-178C objectives for MC/DC coverage.

Lessons Learned

1. Iterative refinements mitigate early risks:
   Iterative verification cycles proved vital in uncovering latent issues early. For instance, addressing the *InadequateCollisionThreatHandling* assertion during the requirements phase prevented logical inconsistencies from propagating to later stages, highlighting the importance of early-stage corrections in reducing downstream rework.

2. Automation enhances certification readiness:
   Automated tools, particularly the LDRA suite, accelerated key compliance activities by streamlining traceability and artifact generation. Notably, automated test generation significantly improved Modified Condition/Decision Coverage (MC/DC) and reduced manual verification workload, though the initial learning curve emphasized the need for dedicated training.

3. Formal methods foster precision:
   The precision offered by formal methods like Alloy and SPIN was pivotal in validating logical consistency and resolving ambiguities across requirements and design. SPIN's validation of liveness properties for collision maneuvers demonstrated the role of model-checking in providing robust safety assurances under diverse operational conditions.

4. Collaboration drives clarity:
   Effective collaboration between domain experts and developers ensured ambiguities in requirements were resolved efficiently. The shared language provided by formal modelling tools such as Alloy bridged communication gaps, ensuring alignment on safety-critical objectives and fostering a unified understanding of system behaviours.

4.5.4 Practical Implications: Best Practices for DO-178C Compliance

This section consolidates key insights from the methodology and case study implementation into actionable best practices for achieving DO-178C compliance. These practices address both technical and procedural challenges inherent in the development of safety-critical software.

Establish Rigorous Requirements Engineering Practices

The foundational step in DO-178C compliance is developing robust, unambiguous requirements. The methodology demonstrated the importance of iterative refinements during the SRATS and HLR verification phases, employing formal methods like Alloy to uncover latent ambiguities and inconsistencies early. Best practices include:

- Quantifiable requirements: Ensure requirements are measurable with precise tolerances to facilitate unambiguous verification.
- Clear terminology and rationale: Use consistent language and separate explanatory rationale to prevent misinterpretation during implementation and review.

Leverage Automation for Compliance Efficiency

Automated tools, such as the LDRA suite, significantly streamline the verification process and compliance activities. Key strategies include:

- Automated traceability: Use tools like TBmanager to maintain bidirectional traceability across requirements, design, and code. This not only satisfies DO-178C objectives but also reduces manual effort and errors.
- Enhanced structural coverage analysis: Automated test case generation was critical in achieving MC/DC coverage, especially in addressing previously uncovered branches during dynamic analysis.

Integrate Formal Methods for Robust Verification

The use of formal methods such as SPIN for architectural model verification and bounded model checking for resolving design-level ambiguities ensured logical consistency and validated safety properties. These methods should be strategically integrated to:

- Validate critical properties: Analyze liveness, safety, and interaction properties to reinforce system reliability under operational scenarios.
- Ensure early detection of errors: Incorporate formal verification early in the lifecycle to mitigate downstream defects, as evidenced by resolving critical assertions in the prioritization logic.

Balance Modularity and Integration

The modular design of the CAS introduced challenges during the integration phase, particularly due to virtual subsystems in Simulink models. Effective practices include:

- Subsystem validation: Perform independent verification of each module before integration, followed by rigorous validation of interaction flows to ensure consistency with architectural requirements.
- Integration testing automation: Leverage tools capable of automating integration testing to validate interaction scenarios across multiple subsystems.

Continuous Learning and Collaboration

The iterative nature of the methodology revealed the importance of ongoing training and interdisciplinary collaboration:

- Training on specialized tools: Invest in comprehensive training for formal tools and automated suites to overcome initial learning curves, ensuring effective utilization.
- Cross-disciplinary communication: Encourage collaboration between domain experts and developers to align safety-critical objectives with practical implementation constraints.

Prioritize Certification-Ready Artifacts

The methodology underscored the need for generating certification-ready artifacts at every phase. Key practices include:

- Integrated documentation: Produce comprehensive artifacts, such as traceability matrices, requirement refinement logs, and structural coverage reports, ensuring they are aligned with DO-178C objectives.
- Iterative artifact validation: Use verification tools to iteratively validate certification artifacts, addressing discrepancies proactively.

By applying these best practices, organizations can navigate the complexities of DO-178C compliance with greater efficiency and confidence. The integration of formal methods, automation, and iterative refinement into the lifecycle not only ensures compliance but also enhances the safety and reliability of safety-critical software. These practices, validated through

the CAS case study, highlight their scalability and applicability across broader domains in avionics software certification.

4.6 Chapter Summary

This chapter presented a robust methodology that combines formal methods with automated verification tools to address the issues of producing DO-178C-compliant software for UAVs. The methodology improves the efficiency and dependability of safety-critical software development by merging the efficiency and automation capabilities of tools such as the LDRA Tool Suite with the mathematical precision of formal methods. By covering every stage of the software lifecycle, from requirements to integration, the method ensures that DO-178C objectives are methodically met, promoting compliance and lowering the possibility of errors at every stage.

This methodology was successfully applied to a Collision Avoidance System (CAS) for UAVs, demonstrating its efficacy in reaching significant milestones such as MC/DC compliance, rigorous bidirectional traceability, and strong structural and functional validations. Although the methodology showed its worth, obstacles were faced, including the adoption of new tools and the complex learning process linked to formal methods. These challenges not only led to methodological refinements but also highlighted opportunities, which will be explored in more detail in the subsequent sections.

Table 4.3 Summary of Verification and Traceability Activities, Tools, Outputs, and DO-178C Objectives Addressed

Activity	**Tool Used**	**Outcome**	**DO-178C Objective**
SRATS Verification	Alloy	Inconsistencies resolved, refined SRATS	
HLR Verification	Alloy	Logical consistency, compliance with SRATS, refinement of HLRs, defining DHLRs	A3.1, A3.2, A3.4
Design Model Verification	Spin	Verified design models, Logical consistency. Compliance with system constraints	A4.8, A4.9, A4.11
LLR Verification	Alloy	Logical consistency. Compliance with HLRs verified	A4.1, A4.2, A4.4
Code Verification	ESBMC	Verified code	A5.3, A5.6
Automated Static Analysis	TBvision	Code reviewed (Coding standards and quality review)	A5.4
Automated Dynamic Testing	TBvision / TBrun	Validation of runtime behaviour (Code coverage and MC/DC planner)	A5.3
Traceability Management (SRATS-HLRs)	TBmanager	Generated traceability matrices. Ensured bidirectional traceability	A3.6
Traceability Management (HLRs-LLRs)	TBmanager	Maintained traceability from HLRs to LLRs. Consistent updates across the lifecycle	A4.6
Traceability Management (LLRs-Code)	TBmanager	Maintained traceability from LLRs to code functions	A5.5
Peer Review	Review Checklists	Review records, resolved issues	A5.1, A5.2

CHAPTER 5 REDAML: A MODELLING LANGUAGE FOR DO-178C HIGH-LEVEL REQUIREMENTS IN AIRSPACE SYSTEMS

5.1 Chapter Overview

This chapter[1] presents the development of ReDaML, a domain-specific modelling language. ReDaML was designed to address the complexities of specifying HLR in SCS while adhering to DO-178C guidelines.

ACPS are examples of safety-critical systems that require strict safety protocols to prevent catastrophic consequences. The DO-178C standard emphasizes the importance of HLRs as the foundation of subsequent software development processes. However, existing approaches to HLR specification often fall short in balancing precision, clarity, and accessibility for diverse stakeholders. These shortcomings can lead to ambiguities, misinterpretations, and inefficiencies, particularly in domains where safety is paramount.

ReDaML was developed to bridge this gap by offering a systematic and organized approach to HLR specification designed especially for safety-critical domains. Its domain-specific modelling approach ensures that safety-related requirements are represented in a precise, verifiable, and analyzable manner. This feature enhances cooperation between technical and non-technical stakeholders while improving the traceability and quality of requirements.

The development of ReDaML was guided by iterative feedback and insights gathered from industrial partners. Initial versions of the metamodel and constraints were shaped by an analysis of existing industrial requirement documents and DO-178C information. Throughout its evolution, ReDaML incorporated feedback from end-users, which influenced refinements such as constraint validation rules, terminology alignment with engineering practice, and mainly the user interface.

The following sections offer a thorough description of ReDaML's architecture, capabilities for modelling safety requirements, and alignment with the DO-178C methodology. Furthermore, a case study of an UAS collision avoidance system is used to illustrate the usefulness of ReDaML in solving real-world safety-critical problems.

[1]Part of the content of this chapter is published in: Misson, H. A., Zrelli, R., Ben Attia, M., Magalhaes, F. G., & Nicolescu, G. (2023, September). ReDaML: A Modeling Language for DO-178C High-Level Requirements in Airspace Systems. In Proceedings of the 34th International Workshop on Rapid System Prototyping (pp. 1-7).

5.2 ReDaML

This section presents ReDaML, a new DSML for requirements data specification based on DO-178C. The modelling language aims to improve the capture of those HLRs related to safety, allowing better analysis and communication between the various disciplines in ACPS software development. Although it was designed with airborne systems in mind, the language can also be used for other critical systems.

The DSML was created using two approaches: 1) studying and analyzing the DO-178C in order to develop a standard-compliant language, and 2) researching currently existing languages in this context. Three UML profiles, as previously noted, served as the foundation for ReDaML: SpecML [68], [45], and SafeUML [69, 70].

Requirements data is the output generated from the activities performed in the software requirements process which has SRATS and system architecture as input. HLRs and DHLRs (if applicable) are part of the output which includes functional, safety-related, and interface requirements.

ReDaML not only gives a way to express these requirements but also enforces DO-178C critical information and objectives that must be met in order to adhere to the standard. This information has been translated into 13 requirements that we judge to be the most expressive and important when it comes to HLR specification based on the standards and analyses made. These requirements are summarized in Table 5.1.

The development is split into three parts: (1) the abstract syntax, which portrays the domain-specific concepts and relations through the metamodel; (2) the concrete syntax, which represents how these concepts are experienced by the user; and (3) the semantic part, which aims to provide domain-specific rules and conforming constraints.

5.2.1 Abstract syntax

Abstract syntax specifies the structure of a language by defining important domain concepts, their relationships, and constraints using a set of language constructs. These constructs are represented in DSML via metamodels. [90] provided the methodology used in the design of the metamodel, which consists of five major steps.

The first step consisted of analyzing, identifying, and collecting relevant information and concepts in the domain. In our case, we examined software HLR specifications considering the DO-178C highlighting safety-related requirements. A glossary of terms[2] was created from this information gathering to organize all possible concepts and terminologies specific to the

Table 5.1 Requirements for ReDaML.

ID	Description
REQ1	The modelling language (ML) shall be able to specify software HLRs, including functional, performance, interface, and safety-related requirements (Section 5.1, 5.1.1, and 11.9 in [3]).
REQ2	The ML shall be able to specify software DHLRs requirements including functional, performance, interface, and safety-related requirements (Section 5.1, 5.1.1, and 11.9 in [3]).
REQ3	The ML shall be able to model traceability from high-level requirements to SRATS (Section 5.1.2, and A-3.6 in [3]).
REQ4	The ML shall be able to model rationale (Section 5.1.2 in [3]).
REQ5	The ML shall be able to model and justify deviations from the Software Requirements Standards (Section 6.3.1 in [3]).
REQ6	The ML shall be able to specify the criticality level of the software, and safety-critical elements (Section 2.3.3 in [3], and IREQ26 in [70]).
REQ7	The ML shall be able to identify safety context in which safety elements are part (IREQ1 in [70]).
REQ8	The ML shall be able to model safety-critical elements (IREQ25 in [70]).
REQ9	The ML shall be able to model software partition for safety-critical elements (Section 11.9 in [3], and IREQ27 in [70]).
REQ10	The ML shall be able to model safety monitoring software (Section 11.9 in [3], and IREQ30 in [70]).
REQ11	The ML shall be able to model safety interfaces for safety-critical elements (Section 11.9 in [3]).
REQ12	The ML shall be able to model software events (Section 2.3.1 in [3], IREQ20, AND IREQ21 in [70]).
REQ13	The ML shall be able to model software interface with other components (Section 5.1.2, and 11.9 in [3]).

area being worked on (such as software level, rationale, traceability, and derived requirements, among others) and to aid in the evolution of subsequent phases.

The second phase involved reviewing the built-in dictionary of terms to confirm the applicability and importance of all the concepts in the emerging language. Usually, when a DSML is established, it is required to refine the information targeting just those that are used to avoid creating potential conflicts and to avoid making the suggested domain too vast [90].

The metamodelling tool is chosen in the third phase. In our case, the tool of choice was

Eclipse Sirius[2], which offers a user-friendly architecture and in-depth documentation. It has become more popular in DSML development and has been adopted by large companies[1].

Figure 5.1 displays the metamodel's implementation (*A complete version can be found in Appendix B*), which is the fourth phase. It depicts a condensed class diagram, where each class stands for a concept that was gathered during the research. These concepts can be further broken down into the six primary categories identified (described below) that meet the requirements chosen in Table 5.1. As seen in the diagram, abstract classes (represented in light grey) are employed, allowing for future metamodel additions.

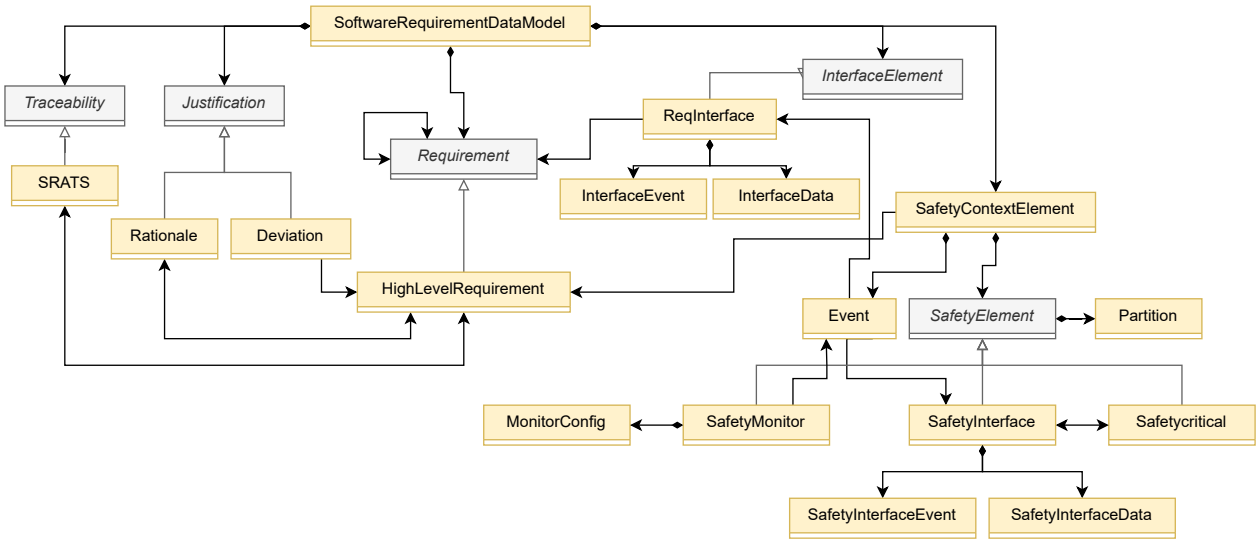

Figure 5.1 Simplified metamodel of ReDaML [91]. It shows the elements and their relationships of an HLR specification for critical systems based on the DO-178C standard.

The modelling language's objective is to enable users to specify and elaborate HLRs and DHLRs based on DO-178C. The collection of these requirements is referred to in the standard as Requirements Data. This essential concept in our approach to software requirements is represented by the *SoftwareRequirementDataModel* (SRDM). It is the base upon which all other concepts are built.

One of the main concepts of our approach is the *Requirement* (abstract) and *HighLevelRequirement* that inherit the attributes of the abstract class. The HLRs and DHRLs are modelled and specified using this concept. The purpose of the specified attributes is to assist the DO-178C-compliant analysis. These ideas meet the requirements in Table 5.1 for REQ1, REQ2, and REQ6.

Traceability is captured by the abstract entity *Traceability* which has *SRATS* as its child.

[2]`https://www.eclipse.org/sirius/`

It is essential to evaluate whether these HLRs can be linked back to SRATS since system requirements were used as input for the generation of the HLRs at this stage of software development. REQ3 is met by applying this concept.

Another key standard-based concept is to provide a rationale for every decision made, which will aid regulatory bodies during the certification process. To achieve the REQ4 goal, we created the abstract class *Justification*, which has two children: *Rationale*, which allows justifying the need for DHLRs, and *Deviation*, which allows justifying any deviation from the Requirements Standard and from the initial project planning.

The *InterfaceElement* concept aims to detail the software interfaces described in interface requirements and is applied to fulfill REQ13. This abstract entity contains the *ReqInterface* class with attributes detailing the interface which is composed of two interface possibilities, *InterfaceEvent* representing interfaces that communicate via asynchronous events, and *InterfaceData* that communicate via data.

The *SafetyContextElement* (SCE) concept incorporates safety features into our language, allowing the user to describe important information that affects the safety of the system in which the software is embedded. By detailing this information, additional resources are provided so that engineers can assess and analyze its scope and propose more effective actions to contain it. Components of a safety-critical nature can be modelled in the context of SCE. There are three types of elements in the *SafetyElements* class: 1) *SafetyCritical* (SC) serves to characterize a software component or function that is isolated by partitioning mechanisms (represented by the *Partition* entity) and to which a software level can be assigned individually (independent of the software system); 2) *SafetyInterface* allows communication between and with the safety component; and 3) *SafetyMonitor*, which indicates the choice of the engineers for implementing monitoring mechanisms. Finally, the *Event* class in SCE specifies events that are triggered in the context of SC and can be propagated through interfaces and may contribute to or control potential hazards outlined in the Safety Assessment Process [92]. REQ6-REQ12 are covered in this instance.

The fifth step is to evaluate the metamodel once it has been specified. A case study was undertaken in collaboration with an industrial partner to identify potential enhancements or adjustments. To facilitate the company's adoption of DSML, a more intuitive layout design is required to explain the concepts to the users in a clearer way. This is a phase of concrete syntax.

5.2.2 Concrete syntax

In the concrete syntax phase, the graphical notation that reflects the notions modelled in the abstract syntax is performed. This acts as a "bridge" between the user and the concepts represented in the metamodel's classes and is essential for DSML comprehension and usability. The authors of [90] mentioned at least two difficulties: the first is a lack of a theoretical foundation for building the design, and the second is that language specification specialists typically focus on abstract syntax and semantics precisely because they lack sufficient knowledge of graphical notation design.

In order to build a concrete syntax, Sirius provides rich tools for specifying representations, which can be in the form of diagrams, tables, matrices, or hierarchies. For the development of these representations, the Viewpoint Specification Model is created, which aims to describe the structure, appearance, and behaviour of the modellers. Figure 5.2 shows the palette with the available elements according to the metamodel. The specification project of concrete syntax can be found in *Appendix B*.

5.2.3 Semantics

In this section, rules and well-formedness constraints will be presented in the form of *Object Constraint Language* (or OCL, which is a formal language for expressing constraints and queries regarding UML models [93]) to enforce the DO-178C information to satisfy its objectives and validate characteristics of the modelling of the Requirements Data.

Domain concepts were wrapped in the abstract syntax by entities in the class diagram, and interactions between these classes, as well as their multiplicities, were specified. However, only these rules and definitions are not able to express more complex and sophisticated constraints by reasoning about class attributes, for example. To this end, OCL enables the assessment and comparison of these attributes and their values defined in the object models (model instances conforming to metamodel), as well as the relationships between classes, using expressions specified in a specific context (the target class). In our context, several expressions have been defined to help engineers ensure mandatory DO-178C objectives. Some of these expressions are depicted in Table 5.2. The comprehensive list of rules can be found in *Appendix B*.

Rule R1 is intended to enforce traceability between specified HLRs and the originating SRATS(s). It is intended to assist the user with respect to the criteria established in the standard, whereby traceability between artifacts of software development is fundamental, even being one of the objectives defined at the output of the software requirements process

Table 5.2 Semantic validation rules using OCL constraints.

ID	Expression Name	Description
R1	SratsTraceability	Traceability between HLRs and SRATS (Section 5.1.2, and A3-6 in [3])
R2	DhlrRationale	Rationale should be provided for DHLR (Section 5.1.2 in [3])
R3	DalValidation	Validation of software level between all components (Section 2.3 in [3])
R4	SafetyCriticalPartition	Safety-critical elements must be isolated from other software components (Section 2.3, and 2.4.1 in [3])
R5	SafetyHLR	Aspects describing safety-related HLR
R6	InterfaceHLR	Aspects describing interface HLR

in that it says *"High-level requirements are traceable to system requirements"* (A-3.6). The rule level defined is *Warning*, in which if the user does not define one or more traces linked to the HLR model, a message is shown (see Figure 5.3).

"Derived high-level requirements and the reason for their existence should be defined", according to DO-178C. The second rule in Table 5.2 is designed to satisfy the statement by alerting the user that when expressing a derived high-level requirement, a rationale must be provided.

The purpose of Rule R3 is to compare the criticality levels of software components. DO-178C allows DAL levels to be identified independently on a component that is isolated by a partition (partitioning between components must be demonstrated). In this context, R3 ensures that the DAL level of the software system as a whole is greater than or equal to the DAL levels of all its component levels.

By analyzing the software components that can impact the safety of the system, characterized by the concepts *SafetyContextElement* and *SafetyCritical* according to the metamodel, these must be isolated from the other components in order to ensure that if an undesired behaviour happens, the rest of the software is protected. This is a fundamental mechanism to ensure that critical components do not affect the system. Rule R4 is intended to alert the user that when a safety-critical element is modelled, partitioning is required.

Finally, R5 and R6 are part of the modelling language design and aim to detail safety-related and interface requirements respectively. The user when specifying requirements of these types should provide more information about what is being specified using the established concepts.

5.3 Application Scenario

An application scenario is offered to demonstrate the practicality of the presented approach and how ReDaML can be used in software development in ACPS. The application case chosen was a collision avoidance (CA) system for an autonomous unmanned aerial system (UAS), which will be discussed further below. The choice of UAS was influenced by its popularity and range of applications, as well as the fact that the industrial partner implements applications in that sector. In the *Appendix B*, the complete diagram, table, and tree of the model, as well as the list of requirements employed, can be found.

5.3.1 Collision Avoidance System Overview

The CA system is critical in autonomous flying as it allows the vehicle to navigate the national airspace system while avoiding hazards and performing its missions and operations successfully. CA's purpose is to provide collision avoidance maneuvers based on data and information collected from various systems and devices. A maneuver is calculated and issued as a command to the flight management system if a potential intruder object is spotted by invading the space defined surrounding the vehicle and could cause a collision.

The CA system primarily gathers data from sensors (such as position, velocity, attitude, and possible invading objects) and the mission management system, which anticipates the next waypoint to be reached. Six major functions are responsible for detecting, tracking, analyzing, and prioritizing potential dangers, as well as determining and ordering the estimated maneuver [94]. All details and requirements considered in this scenario were also obtained from this same source.

5.3.2 Modelling with ReDaML

To model the HLRs using ReDaML, all requirements process activities must first be completed. Based on the SRATS, system architecture, and safety assessment obtained following [94, 95], 39 HLRs and DHLRs were identified from these process activities. Due to space constraints, only HLR-4 and its derivation DHLR-4.1 are discussed.

HLR-4 *(Functional): The CA subsystem software shall assign a priority level to each tracked traffic element based on the assessed collision threat.*

DHLR-4.1 *(Safety): The CA subsystem software shall prioritize every traffic element that has been deemed a collision threat.*

Modelling requirements data is intuitive, and adding the HLRs can be done either through

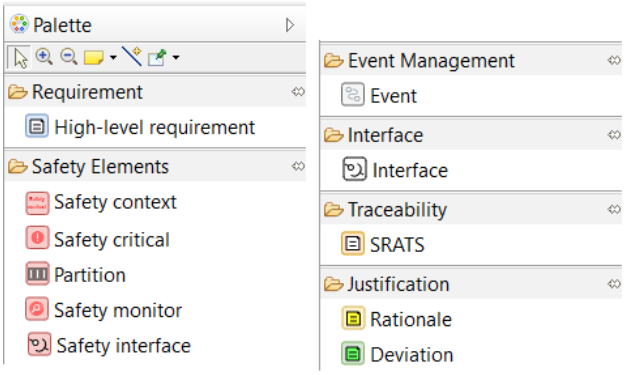

Figure 5.2 Modelling tool palette.

the palette in the diagram or by using the table feature. However, the other artifacts that make up the model (such as rationale, requirements details, and safety context) must be included in the diagram.

Figure 5.4 depicts a portion of the diagram illustrating the HLR-4 and DHLR-4.1 specifications. ReDaML has a distinct advantage over other languages in its capacity to describe requirements. The combination of elements enables engineers to specify requirements in an efficient and comprehensive manner. Furthermore, the tool uses intuitive colour schemes to draw attention to important information and improve comprehension. Typically, HLRs are composed of elements that tie everything together by revealing important considerations. To this end, ReDaML enforces this composition through links that represent both details that allow a better understanding of the requirement and provide rationale and traceability, which are relevant points covered by DO-178C. Examples of this composition can be seen in Figure 4, where the dotted line called *traces* represents traceability with system requirements, derived requirements, such as DHLR-4.1, are linked to the HLR-4 from which they were formed through a solid line called *justifies*.

One of ReDaML's advantages over other existing methods is the ability to provide a level of detail on safety and interface requirements. This ability aids the software development process in that it enables the concerns that must be taken into account while creating the architecture and low-level requirements to be made apparent and explicit. Figure 4 provides an illustration of this degree of information. The safety context is represented by the rectangle on the left, which is connected to DHLR-4.1 by the dashed line labelled "describes". Important aspects that should be considered throughout the design process and that can contribute to the safety of the system are presented in this context. This combination is a valuable asset enabling a deeper analysis of HLRs while allowing engineers to provide helpful insights. In addition, according to the standard, details of derived requirements must be transmitted

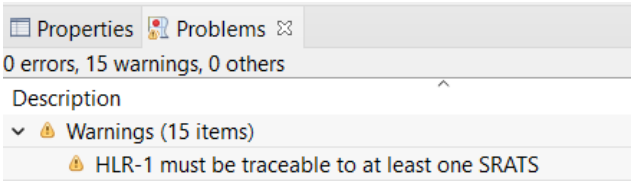

Figure 5.3 Warning message from R1 rule.

to system engineers to assess safety impacts. Considering the level of detail provided by ReDaML can contribute to this aspect of the process, which is not seen in other UML-based languages.

Using ReDaML to model the requirements of the CA system has various advantages, particularly when compared to traditional methods. Compliance with DO-178C objectives and activities, such as traceability between HLR and SRATS and also between HLR and DHRL, rationale for DHRLs, and verification of compatibility between the constituent components' software levels, is one of the main benefits. Furthermore, the features provided enable engineers to specify, review, and analyze requirements in a less error-prone and time-consuming manner than if done manually. Communication between teams improves since requirements are still stated in natural language, reducing the need for an expert.

Overall, the use of our approach in the development of the case study demonstrated an increase in the quality of the software compared to the traditional methods practiced previously. The precision and coherence it has brought to the specification process are the greatest benefits, even helping in the development of the design.

5.4 Considerations

ReDaML is primarily intended for requirements engineers and software developers involved in the early stages of software development for safety-critical systems. Its purpose is to support the modelling of HLRs in a way that promotes consistency, precision, and traceability. By enforcing a standardized structure for requirement specification, ReDaML mitigates the ambiguities and inconsistencies commonly found in natural language descriptions. This structured modelling approach helps improve the overall quality of requirements, facilitates downstream automation (such as transformation to formal models or architectures), and contributes to reducing rework, development time, and costs, which are critical concerns in the aerospace domain.

Although ReDaML currently does not directly generate formal models, it is designed to

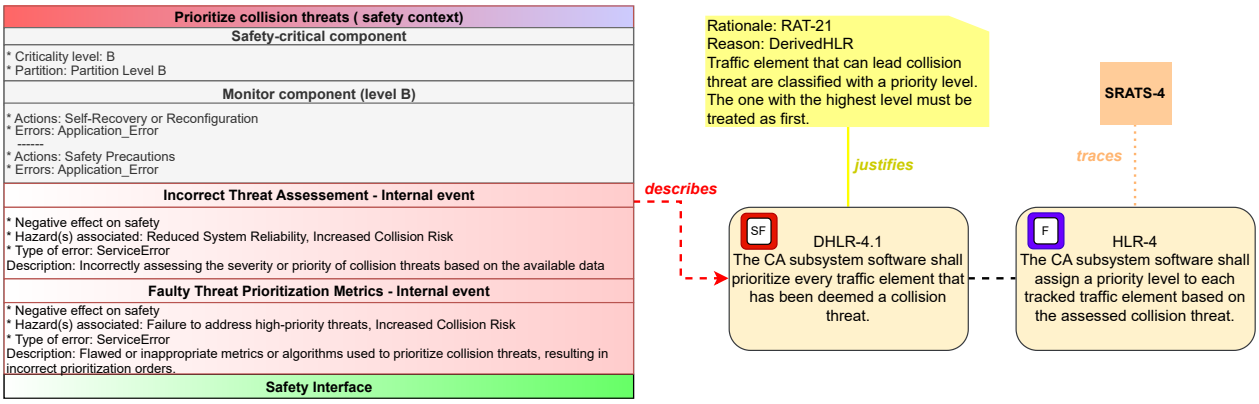

Figure 5.4 HLR-4 and DHLR-4.1 modelled with ReDaML [91] diagram view. It depicts the HLR aspects according to DO-178C. The functional type HLR is traced to SRATS-4 on the right. Additionally, a derived requirement of the safety type related to the HLR is represented, together with a rationale behind it. Given that it is a safety type, ReDaML imposes the description to provide more details.

facilitate model transformations into formal representations through well-defined mapping rules. In particular, a transformation has been implemented to convert validated ReDaML models into AADL architectures (explored in section 6), enabling early analysis of system structure and compliance with ARINC 653 constraints. Additionally, model transformation mechanisms can also be used to generate formal models, such as Alloy. In this approach, a set of mapping rules must be defined to translate ReDaML elements into Alloy constructs, ensuring that the transformation complies with the Alloy metamodel and semantics. This compatibility opens opportunities for formal verification of requirements, consistency and satisfiability using the Alloy Analyzer.

Moreover, the same transformation approach could be extended to generate temporal logic specifications, such as Linear Temporal Logic (LTL), which are suitable for verifying architectural models via tools like Spin. While such transformations are not yet implemented, ReDaML's modular metamodel provides a sound foundation for supporting multiple formal backends in the future, making it a versatile asset for model-based development and formal assurance.

5.5 Chapter Summary

This chapter introduced ReDaML, a domain-specific modelling language developed to address the challenges of specifying high-level requirements for safety-critical software systems in alignment with DO-178C guidelines. By emphasizing clarity, precision, and the integra-

tion of safety principles, ReDaML enhances the quality of requirements specifications while facilitating thorough analysis throughout the development process. The model representing HLR ensures that key safety objectives are met and provides a structured foundation for subsequent stages of software development. To demonstrate its practicality and effectiveness, a case study was conducted, showcasing the application of ReDaML and its ability to support the development of DO-178C-compliant systems.

CHAPTER 6 BRIDGING THE GAP: A DO-178C COMPLIANT FRAMEWORK FOR REQUIREMENTS-TO-ARCHITECTURE TRANSITION

6.1 Chapter Overview

The software architecture plays a pivotal role in developing safety-critical systems, acting as a blueprint that ensures consistency and alignment between system requirements and implementation. Beyond its technical utility, it facilitates stakeholder communication, promotes mutual understanding, and provides developers with a transferable, reusable, and implementation-independent representation of the system. Nevertheless, there are multiple challenges in developing software architectures, especially concerning DO-178C compliance. One of the most critical challenges is ensuring traceability and completeness across all artifacts throughout the software lifecycle, encompassing both functional and non-functional requirements. Neglecting these aspects can lead to costly errors or catastrophic failures, making it imperative to develop systematic approaches for translating high-level requirements into a cohesive and compliant software architecture.

This chapter [1] addresses the challenges of transitioning from HLR to software architecture in safety-critical systems. It highlights the complexity of capturing the entirety of requirements while ensuring consistency and compliance with DO-178C standards. By identifying gaps in existing methodologies and practices, the chapter introduces a structured framework that leverages model transformation mechanisms to automate and streamline this transition. This approach not only reduces the potential for errors but also ensures that both functional and non-functional requirements are seamlessly integrated into the architecture, enhancing the overall quality of the system.

This work uses model transformation techniques to make the process of developing software architectures more robust and accessible, even for engineers with varying levels of expertise. By automating key aspects of the transition from HLR to architecture, the proposed methodology supports the creation of high-quality, compliant artifacts while reducing development time and effort. The contributions presented in this chapter are intended to address the pressing challenges of software development in safety-critical domains, ensuring reliability, traceability, and adherence to stringent industry standards.

[1] Misson, H.A., Zrelli, R., Ben Attia, M., Magalhaes, F.G.d., Shabah, A., & Nicolescu, G. (2025). Bridging the Gap: A DO-178C Compliant Framework for Requirements-to-Architecture Transition. Submitted to the Aerospace Systems journal.

6.2 Framework Overview: Bridging Requirements to Architecture

Developing software compliant with standards like DO-178C presents significant challenges, particularly in seamlessly integrating requirements with software design while ensuring safety [7]. A primary issue is developing effective methodologies for deriving software architectures directly from HLRs, a task complicated by the need to incorporate safety-critical characteristics. This process often relies on the expertise and intuition of software engineers, making structured methodologies crucial for less experienced practitioners [72].

Our framework streamlines the complex process of translating HLRs into a software architecture aligned with avionics standards. The objective is to enhance the efficiency and reliability of safety-critical systems development. Figure 6.1 illustrates the framework's workflow within the DO-178C defined software development process. While the core transition between the requirement and design is emphasized, our framework extends beyond this by encompassing these two processes. The left side of the figure represents standard DO-178C processes, while the right side, enclosed within a dotted line, outlines our proposed methodology. Our methodology begins with the modelling process, where a structured requirements data model is generated. As the second step, the requirements undergo a thorough review and verification phase to ensure compliance with DO-178C. This step guarantees that the requirements remain consistent, unambiguous, and accurate, aligning with industry certification standards. The final stage is model transformation, which uses the requirements data model to generate an AADL architectural model that adheres to HLRs.

The first stage involves modelling high-level requirements using ReDaML [91], a modelling language that accurately captures requirements while facilitating compliance with regulatory standards. ReDaML is particularly effective for modelling and reviewing requirement specifications since it enforces the actions and objectives outlined in DO-178C, ensuring completeness and regulatory compliance.

Verification is a fundamental process throughout the software lifecycle [3]. Its primary goal is identifying issues early, preventing them from leading to faults or failures. The verification process includes reviews, analyses, and tests. Reviews and analyses assess the accuracy, completeness, and verifiability of the lifecycle outputs, such as the Requirement Data. While reviews provide qualitative assessments, analyses offer consistent evidence to ensure correctness [7].

The model transformation process begins once HLRs have been modelled and validated. The software architecture is then built using an automatically generated AADL model derived from the validated ReDaML model. This architecture follows an ARINC 653 pattern model

tailored explicitly for avionics systems, ensuring compliance with industry standards and best practices. By adopting an ARINC 653-based architecture, the framework enhances software safety and reliability through time and space partitioning mechanisms that prevent error propagation between partitions [14].

The transformation process is based on the ReDaML and AADL metamodels, enabling smooth conversion while preserving the integrity of the original requirements. Transformation rules established within the framework ensure that the generated architecture remains traceable to the high-level requirements, guaranteeing conformance and completeness. This approach considerably reduces the likelihood of syntactic and semantic errors, as the architecture is inherently tied to the initial requirements.

Furthermore, the resulting architecture not only fulfills DO-178C objectives but also establishes a comprehensive mapping of HLR to architectural components. This alignment guarantees that all high-level requirements are addressed within the architecture, supporting traceability and accountability throughout the development lifecycle. By embedding DO-178C concepts directly into the design, the framework minimizes risks while improving the overall quality and reliability of the software system.

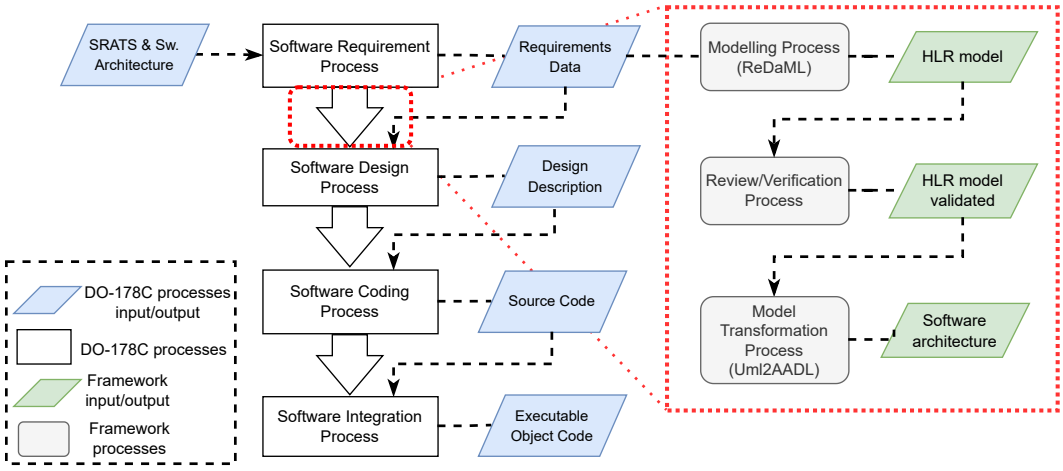

Figure 6.1 Workflow of the proposed Framework within the dotted rectangle and its relationship with DO-178C processes. It highlights the HLR modelling and verification, in which its model serves as the input of the model transformation to generate the architecture.

6.2.1 Methodology

The development of the framework for transitioning from HLRs to software architecture follows a structured methodology designed to address the specific challenges of safety-critical

software development in avionics. A comprehensive literature analysis provided the foundation for this approach, incorporating insights into current architectural practices and their alignment with Do-178C objectives and activities.

Identifying the challenges in transitioning from HLRs to software architecture was a crucial step. This knowledge guided the choices and steps made during the approach to ensure a smooth and compliant translation process.

Suitable modelling languages were assessed to represent software architecture and HLRs accurately. AADL was selected for architecture representation due to its compatibility with avionics systems. At the same time, ReDaML was chosen to model HLRs because of its ability to capture safety-critical requirements in compliance with DO-178C.

The selection of a transformation engine was pivotal to facilitate the conversion from the DSML to AADL. Acceleo M2T[2] was selected due to its open-source nature, integration with the Eclipse IDE, support for OMG MOF [96] standards, and advanced capabilities such as syntax highlighting, error detection, and refactoring[3]. This selection ensured efficient and accurate transformations while maintaining specification integrity.

Another important step was defining the transformation rules. These rules, based on the meta-models of both the source DSML and the target AADL, ensured DO-178C compliance and preserved specification integrity throughout the process. Thorough testing and validation validated the transformation framework's accuracy and reliability.

Finally, we applied an iterative approach to continuously refine the framework based on feedback, experience, and evolving avionics software development requirements. This iterative process guaranteed that the framework remained optimized and adaptable to changing needs.

6.2.2 Model Transformation

An MDA-compliant methodology is used to translate high-level UML requirements into concrete software architecture, with model transformation serving as a key component. This process bridges the gap between Requirements Data captured using the DSML and architectural components represented in AADL. By automating model conversion, the transformation process streamlines development, enhances consistency, and ensures adherence to DO-178C standards.

Figure 6.2 provides an overview of the transformation process. It operates on source and target models, serving as input and output for the transformation engine. Each model

[2]https://eclipse.dev/acceleo/
[3]https://eclipse.dev/acceleo/resources.html

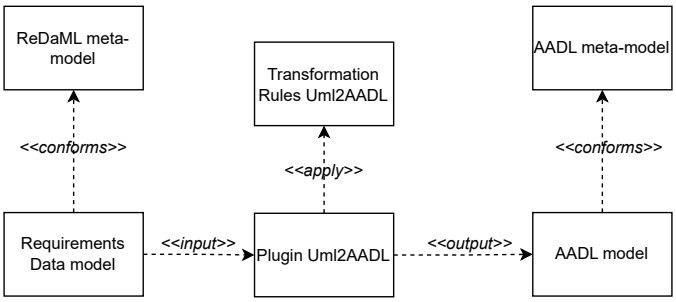

Figure 6.2 Structure of the model transformation process.

adheres to its respective metamodels, while the transformation engine follows predefined transformation rules.

A set of transformation rules has been established to guide this process, mapping ReDaML metamodel elements to AADL metamodel elements. These rules have been implemented in a prototype tool named ReDaML2AADL. Based on the partner requirements, an ARINC 653 model supporting IMA was designed to comply with avionics development standards and avoid excessive generalization of the resulting architecture.

Each model developed using ReDaML corresponds to a component of the software system under development. Consequently, each generated architecture aligns with the corresponding component's structure and its interactions with other software or hardware components within the system.

Domain-specific Modelling Language

The requirements process stands as a cornerstone of the software development lifecycle. A project's success or failure is closely tied to the quality and clarity of its requirements [7]. In safety-critical systems, strict safety standards must be followed, highlighting the significance of stringent requirements procedures. These processes serve as the initial step in identifying and mitigating potential hazards and risks within the software system. By rigorously defining and documenting requirements, developers establish a solid foundation for building safety-critical software, ensuring that safety considerations are incorporated throughout development [97].

Given the challenges of specifying requirements in avionics and the need to incorporate safety-related features, a systematic approach is required. Using a domain-specific language provides a structured and formal means to assess, test and verify selected requirements. [63].

ReDaML [91] is employed to improve the capture of safety-related HLRs, allowing for im-

proved analysis and communication across disciplines in Airspace Cyber-Physical Systems (ACPS) software development. Although originally designed for airborne systems, ReDaML can be applied to other critical systems.

Figure 5.1 illustrates an overview of its metamodel, depicted as a simplified class diagram without explicit attributes. Each class represents a concept based on avionics standards. The modelling language allows users to specify and develop HLRs and Derived HLRs (DHLRs) in alignment with DO-178C, which refers to these requirements collectively as "Requirements Data".

Compliance with DO-178C is embedded within the abstract syntax that defines the language. This includes a rationale for justifying derived requirements, traceability between system requirements and software HLRs, and attributes describing each HLR, such as whether the requirement specifies design details at a granular level.

Another aspect of compliance is the semantic component of the language, which provides OCL-based rules [93] for validating models. For example, when a requirement is designated as derived, meaning it does not originate from a higher-level requirement but supplements the behaviour of the software. In this case, a rule mandates that the user provide a justification. This justification is required for the system team to analyze its impact during development. These rules and guidelines are derived from an analysis of the DO-178C document, referenced in [91].

This phase contributes to software development by enabling engineers to collect higher-quality requirements compared to traditional manual specifications methods. This improvement stems from the formalization of domain-specific concepts within the language, which ensures greater accuracy in requirement expression. Furthermore, verifying the model against predefined constraints enhances precision.

ReDaML is used to create the requirements model, which allows the high-level requirements to be accurately represented. This model also captures necessary information and serves as input for the model transformation process, which is detailed below.

AADL ARINC653 model

A template model was developed for the architecture to support ARINC 653 and verify compliance with DO-178C objectives. This template plays a critical role in ensuring that software architecture aligns with the stringent safety and certification standards in avionics systems. By incorporating ARINC 653 features such as partitioning and Health Monitoring (HM), the template provides a robust foundation for developing systems that meet high-level

safety requirements. Its design facilitates the seamless integration of these features, enabling efficient and accurate DO-178C verification compliance throughout the software development lifecycle.

The template is designed for adaptability, allowing our partners to customize it based on specific hardware and communication systems while maintaining the integrity of ARINC 653 features and safety-related elements from the HLR model. Employing model transformation, the template generates essential architectural components like partitioning and HM, guaranteeing that these safety measures are applied accurately and consistently across multiple projects.

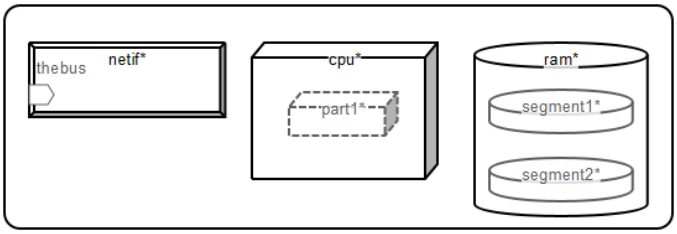

Figure 6.3 AADL components representing network interface containing the bus (in the left), processor with one partition (in the center), and memory with two segments (in the right)

The AADL elements utilized in the template model for processor, memory, and communication resources are shown in Figure 6.3 and detailed in Table 6.1.

The generated software architecture comprises processes that execute code segments (threads) within each partition. To ensure correct execution and resource management, each process is associated with a virtual processor and a memory segment. This association is achieved using the *Actual_Processor_Binding* property in AADL. This property links each process to a specific virtual processor, enforcing ARINC 653 time partitioning and real-time scheduling.

Actual_Memory_Binding assigns each process to a dedicated memory segment, ensuring compliance with ARINC 653 space partitioning, preventing cross-partition interference, and improving system reliability.

In our template, we also employed HM, a critical feature for avionics software robustness and reliability. HM mechanism in ARINC 653 detects and addresses faults at multiple levels, including module, partition, and process, by defining appropriate responses to minimize impact and enhance safety.

The AADL ARINC653 library seamlessly integrates HM functionalities, utilizing constructs such as *HM_Errors* and *HM_Actions* to define fault detection and mitigation strategies.

Table 6.1 Details of AADL elements for representing processor, memory and communication resources in template model.

Component	Description
Network Interface and Bus	**Network Interface**: Manages system communication, ensuring efficient data transmission and reception across avionics components. **Bus**: Works alongside the network interface to establish communication pathways, enabling the network interface to function effectively.
Memory	**RAM**: Provides the volatile memory necessary for software execution. **Segment Memories**: Defined within RAM, these segments are crucial for space partitioning as per ARINC 653 requirements. They ensure that different applications and processes operate in designated memory regions, preventing interference and ensuring stability.
Processor	**Processor**: Executes software applications and manages avionics software operations. **Virtual Processor**: Supports ARINC 653 time partitioning. Integrated within the main processor, each virtual processor is associated with a major timeframe, defining execution time slots for different partitions. This mechanism guarantees scheduled execution, deterministic behavior, and compliance with real-time requirements.

ReDaML2AADL

This section details the mapping between ReDaML and AADL, which serves as the foundation for transformation rules in ReDaML2AADL. This mapping is essential for translating software component requirements data into a corresponding architectural model.

The transformation from ReDaML to AADL is not a simple one-to-one mapping, where each ReDaML class directly corresponds to an AADL construct. Instead, transformation rules are defined based on the semantics of ReDaML classes and attributes. This semantic-driven method ensures that the architectural model accurately represents the ReDaML model's requirements and constraints.

The generated architecture provides two perspectives: one depicting the application with its processes and interrelations and another providing a system-level view that highlights

interactions between the software and other applications or equipment. Since ReDaML was designed for modelling individual software components, the resulting architecture concentrates on describing a single component within a larger system.

When modelling the HLR in ReDaML, an AADL system type and system implementation are created, along with a process representing the software. Additional relationships and components (software or hardware) are structured based on the defined interfaces.

The mapping between source and target metamodel components is outlined as follows:

- **Top-Level System Mapping**:
  - Map the ReDaML top-level system to an AADL system component, representing the overall system structure.
  - Define an AADL system implementation that includes subcomponents (systems, devices, and connections) that reflect the hierarchical organization of the overall system.
- **Main Software Mapping**:
  - Map the main software element in ReDaML to an AADL system component, encapsulating core functionality. Within the AADL system, define data ports, event data ports, and/or event data ports to represent software interfaces.
  - Create an AADL system implementation that includes resources, processes, and connections between system and process boundaries. Additionally, create a process within the AADL system to represent the software's execution context.
- **External Components Mapping**: From the software interfaces, generate corresponding external software and hardware components that are part of the top-level system. If a software component is present, map it to an AADL system component. If a hardware component is present, map it to an AADL device.
- **Annex Error and Behavior Mapping**: If safety context elements exist in the Requirements Data model, incorporate Annex Error and Behavior constructs within the system implementation of the main software. These annexes help address safety constraints and fault-handling mechanisms.

To demonstrate the transformation rules between ReDaML and AADL, Figure 6.4 illustrates an excerpt of the ReDaML metamodel. The algorithm for this conversion, along with the expected AADL output, will be presented.

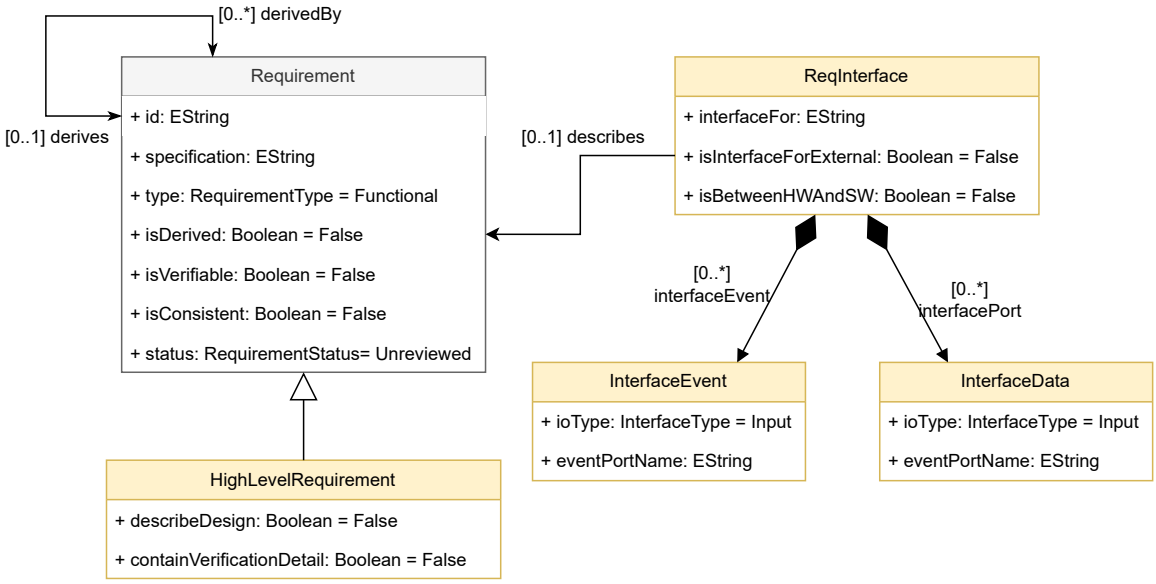

Figure 6.4 Extraction of ReDaML metamodel [91]. It considers elements that model HLRs and their relationships with other components through interfaces. For example, a requirement specifying that the application must send or receive data from another application or piece of equipment. When this interaction is through sampling ports, i.e. the data must be fresh, the "*InterfaceData*" element is modelled. If it is an occasional interaction, via a queuing port, the element representing "*InterfaceEvent*" is modelled.

One transformation rule specifies how to generate an architecture component for a software application that interfaces with another application and its features in AADL. Table 6.2 outlines the conditions for this transformation.

Table 6.2 Transformation rule *Create_External_App* with pre and post conditions

Rule Name	**Create_External_App**
Preconditions	1 - $Interface_{source}$ is an instance of the ReqInterface class in the source metamodel 2 - *isInterfaceWithExternal(*$Interface_{source}$*)* returns true 3 - *isBetweenHWandSW(*$Interface_{source}$*)* returns false
Postconditions	- $System_{target}$ is a new instance of the System component in the AADL model - $Feature_{target}$ are features associated with $System_{target}$

Algorithm 1 Generate AADL Systems from Software Requirement Data Model

1: **procedure** CREATESYSTEM(m : SoftwareRequirementDataModel)
2: **let** reqInterfaces : Set(ReqInterface) = m.eContents(ReqInterface) → select(isInterfaceWithExternal and not isBetweenHWAndSW) → asSet()
3: **let** reqInterfaceForValues : Set(String) = reqInterfaces.interfaceWith → asSet()
4: **for all** interfaceForName : String **in** reqInterfaceForValues **do**
5: **system** interfaceForName
6: **features**
7: **for all** it : ReqInterface **in** m.eContents(ReqInterface) → select(interfaceWith = interfaceForName) **do**
8: it.generateDataPorts()
9: it.generateEventPorts()
10: **end for**
11: **end** interfaceForName
12: **end for**
13: **end procedure**

Algorithm 1 defines this transformation rule in Acceleo M2T. For HLRs with interfaces where *isInterfaceWithExternal* is true and *isBetweenHWAndSW* is false, an AADL system component is created. This component represents the software application interfacing with the specified external software. The use of sets in lines 2 and 3 of the algorithm aims to avoid duplicate system creation. Additionally, system features such as data ports and/or events ports are generated via *generateDataPorts* and *generateEventPorts* templates, leveraging the modular structure of the transformation algorithm.

Listing 6.1 depicts the AADL model developed by the presented algorithm alongside the ReDaML model. It defines the Sensor Fusion application, which interacts with sensor de-

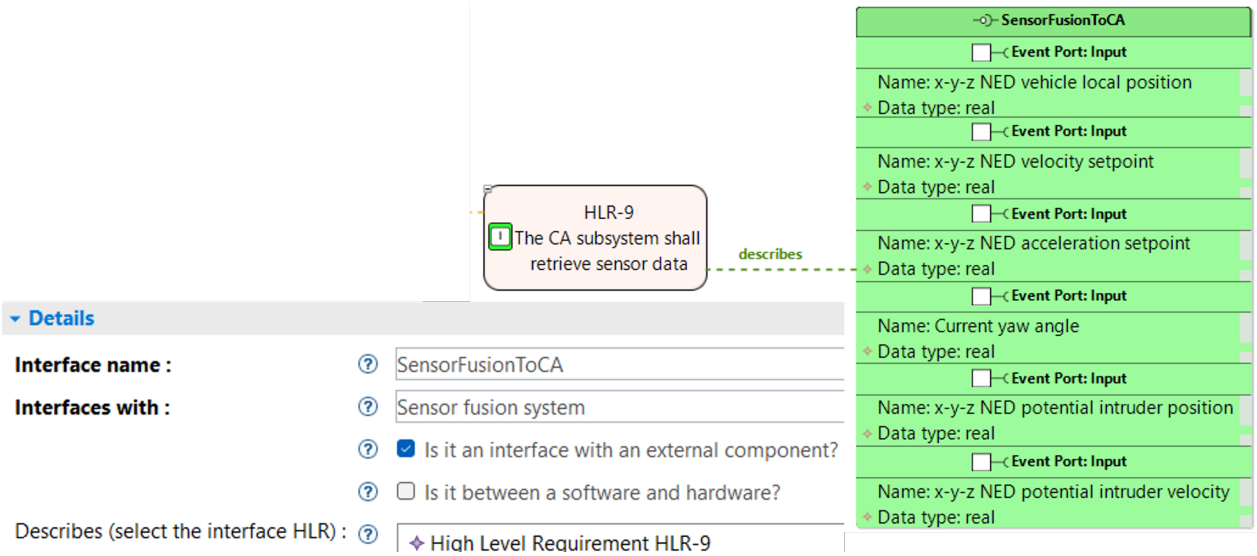

Figure 6.5 Example of HLR with interface description using ReDaML

vices and provides information to the components that require it. The developed software component is Collision Avoidance, which is discussed in the Case Study below. The Sensor Fusion application is not fully described; only its interactions with the Collision Avoidance component are specified.

```
system sensor_fusion_system
  features
    x_y_z_ned_vehicle_local_position_port_out: out data port DataTypes::real;
    x_y_z_ned_velocity_setpoint_port_out: out data port DataTypes::real;
    x_y_z_ned_acceleration_setpoint_port_out: out data port DataTypes::real;
    current_yaw_angle_port_out: out data port DataTypes::real;
    x_y_z_ned_potential_intruder_position_port_out: out data port DataTypes::real;
    x_y_z_ned_potential_intruder_velocity_port_out: out data port DataTypes::real;
end sensor_fusion_system;
```

Listing 6.1 AADL Example: Sensor Fusion System

As depicted in the AADL listing 6.1, the Sensor fusion application is part of a broader system view that includes multiple applications and devices. This higher-level system view illustrates how various components, such as the Sensor Fusion and Collision Avoidance, communicate and interact through their defined ports. Listing 6.2 provides the AADL representation of the Collision Avoidance software, showing its features and interactions.

The transformation algorithm, developed leveraging Acceleo M2T, is structured into five distinct templates, each designed to simplify module development based on mapping criteria. These templates serve as modular building blocks, encapsulating the transformation logic for DSML parts and their corresponding AADL components. This modular design en-

```
system implementation collisionavoidance_system.impl
  subcomponents
    collisionavoidance_sw: system collisionavoidance_pkg::collisionavoidance;
        ...
    sensor_fusion_system_sys: system ExternalComponents::sensor_fusion_system;
  connections
        ...
    C5: port sensor_fusion_system_sys.x_y_z_ned_vehicle_local_position_port_out ->
        collisionavoidance_sw.x_y_z_ned_vehicle_local_position_port_in;
    C6: port sensor_fusion_system_sys.x_y_z_ned_velocity_setpoint_port_out ->
        collisionavoidance_sw.x_y_z_ned_velocity_setpoint_port_in;
    C7: port sensor_fusion_system_sys.x_y_z_ned_acceleration_setpoint_port_out ->
        collisionavoidance_sw.x_y_z_ned_acceleration_setpoint_port_in;
    C8: port sensor_fusion_system_sys.current_yaw_angle_port_out -> collisionavoidance_sw.
        current_yaw_angle_port_in;
    C9: port sensor_fusion_system_sys.x_y_z_ned_potential_intruder_position_port_out ->
        collisionavoidance_sw.x_y_z_ned_potential_intruder_position_port_in;
    C10: port sensor_fusion_system_sys.x_y_z_ned_potential_intruder_velocity_port_out ->
        collisionavoidance_sw.x_y_z_ned_potential_intruder_velocity_port_in;
     ...
end collisionavoidance_system.impl;
```

Listing 6.2 AADL Example: Sensor Fusion System

sures clarity, maintainability, and extensibility, allowing for efficient adaptation to evolving requirements and mapping specifications.

The collection of transformation rules was validated against multiple scenario instances to ensure correctness. In the provided example, the implemented scenarios cover various combinations of the attributes *isInterfaceWithExternal* and *isBetweenHWandSW*. Table 6.3 summarizes the test results.

Table 6.3 Test cases to validate the correctness of transformation rules. Each scenario was created from an empty initial scenario.

Scenario	**Description**	**Expected Result**
1	Empty scenario (without interfaces)	No system created
2	Add 1 interface with *isInterfaceWithExternal* equals 'true' and *isBetweenHWandSw* equals 'true'	No system created
3	Add 1 interface with *isInterfaceWithExternal* equals 'true' and *isBetweenHWandSw* equals 'false'	Create 1 system
4	Add 1 interface with *isInterfaceWithExternal* equals 'false' and *isBetweenHWandSw* equals 'true'	No system created
5	Add 1 interface with *isInterfaceWithExternal* equals 'false' and *isBetweenHWandSw* equals 'false'	No system created
6	Add 2 interfaces with *isInterfaceWithExternal* equals 'true' and *isBetweenHWandSw* equals 'false'	Create 2 systems
7	Add 3 interfaces with *isInterfaceWithExternal* equals 'true' and *isBetweenHWandSw* equals 'false'	Create 3 systems

AADL defines a 'system' element as a fundamental construct representing a comprehensive

entity within a system architecture. A system element in AADL may consist of software components. It provides a high-level abstraction for modelling large system architectures by specifying the structure, behaviour, and interconnections of multiple components.

The transformation rules were verified by checking whether the correct number of systems were produced based on the interface attributes provided in the ReDaML mode (SoftwareRequirementDataModel):

- **Scenario 1**: When no interfaces are provided, no systems should be created.
- **Scenario 2**: If an interface has both *isInterfaceWithExternal* and *isBetweenHWandSW* set to true, no system should be created.
- **Scenario 3**: An interface with *isInterfaceWithExternal* set to true and *isBetween-HWandSW* set to false should result in the creation of one system.
- **Scenario 4 & 5**: Interfaces where *isInterfaceWithExternal* is false should not result in system creation, regardless of the *isBetweenHWandSW* attribute.
- **Scenario 6 & 7**: Adding multiple interfaces with *isInterfaceWithExternal* set to true and *isBetweenHWandSW* set to false should result in the corresponding number of systems being created.

Executing these tests allows for the evaluation of transformation rules under various conditions. This comprehensive testing ensures that the rules function as intended, validating their correctness.

Acceleo M2T was selected as the model transformation tool to generate textual artifacts from the input model. By creating plugins, transformation logic can be integrated into the Eclipse IDE. These plugins empower users to perform transformations directly in Eclipse, which improves the workflow by facilitating a seamless transition from modelled and reviewed requirements data to the AADL architecture within a familiar environment, ensuring efficiency throughout the development process.

6.3 Case Study

This section aims to illustrate the practical feasibility and efficacy of the framework developed for translating HLR compliant with DO-178C into software architecture. This comprehensive demonstration encompasses the entire framework workflow, starting from the initial

modelling of HLRs using ReDaML, followed by a thorough review process to refine and validate the Requirements Data model. Subsequently, the transformation process is executed to seamlessly transition the refined Requirements Data model into a corresponding software architecture. Figure 6.6 shows the process for the activities covered.

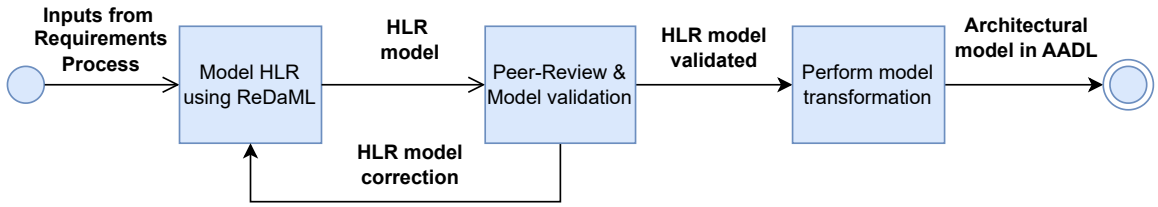

Figure 6.6 Activity diagram illustrating the process of transforming High-Level Requirements (HLRs) into an Architecture model in AADL.

For validating the framework we have chosen as a case study the Collision Avoidance System (CAS) system for an autonomous Unmanned Aerial Vehicle (UAV). In the world of autonomous flying, the CAS system is critical because it allows the vehicle to navigate through the national airspace system while reducing potential hazards and performing missions and operations with precision. Given the importance of safety in autonomous flight, the CA system relies heavily on advanced technology to maintain a degree of flight safety comparable to manned flights.

The major goal of the CAS system is to facilitate collision avoidance manoeuvres by combining data and information from multiple onboard systems and external sensors. These manoeuvres are calculated and implemented as commands to the flight management system when prospective intruder objects enter the specified airspace surrounding the vehicle, preventing collisions. The functional requirements of the CAS system consist of the following major functionalities: Detect Traffic, Track Traffic, Evaluate Collision Potential, Prioritize Collision Threats, Determine Avoidance Maneuver and Command Maneuver.

Below we will present each activity in the workflow in more detail:

Activity 1 - Model HLR using ReDaML The workflow's initial step involves utilizing ReDaML to model the HLR. The collection of HLRs produced throughout the requirements process serves as the input for this task. Every requirement is rigorously modelled guaranteeing that all pertinent information is traceably and structuredly captured. A subset of the requirements data will be presented to demonstrate this procedure, as denoted as follows.

HLR-4 *(Trace to SRATS-4) (Functional Req): The CAS subsystem software shall assign a*

priority level to each tracked traffic element based on the assessed collision threat.

DHLR-4.1 *(Trace to HLR-4) (Safety Req): The CAS subsystem software shall prioritize every traffic element that has been deemed a collision threat.*
Derived Rationale: Traffic elements that can lead to collision threat are classified with a priority level. The one with the highest level must be treated first.

Figure 5.4 depicts the diagram illustrating the HLR-4 and DHLR-4.1 specifications (a comphreensive set of all the requirements used can be found in Appendix B). Normally, HLRs are formed up of elements that connect everything by exposing essential information. To accomplish this, ReDaML enforces this composition through links that indicate details that aid in understanding the requirement, rationale (if applicable) and traceability, all of which are addressed by DO-178C. In the model, the dotted line *traces* indicates traceability to system requirements, whereas the solid line *justifies* links derived requirements, such as DHLR-4.1, to the HLR-4 from which they were created.

Since DHLR-4.1 is a safety requirement, the modelling language requires the user to provide additional information about how this requirement affects the system's safety. To do this, a safety context (big square on the left) must be established, allowing the team to preserve this critical aspect throughout development.

The final CAS software model reflecting the requirements data is comprised of all elements created using the tool. Once all of these elements have been modelled, the next step is to review them to ensure that standard procedures are followed.

Activity 2 - Peer-Review & Model Verification In compliance with DO-178C guidelines, our research verifies requirements through a person who was not engaged in their development or modelling. The necessity of independence in the verification process is mandated by the Design Assurance Level (DAL) B classification of the Collision Avoidance system, considering its potential impact.

In this paper we only concentrate on peer-review processes; testing and other types of verification are not included. To guarantee adherence to internal standards and DO-178C, peer review is necessary. An implemented checklist is followed during the review to ensure that important configuration management and quality components—like version control, baselines, and modification tracking—are met. Model validation is made possible by ReDaML's use of OCL constraints, which facilitate this procedure. This aids reviewers in confirming that the objectives of DO-178C are fulfilled. The validation process is depicted in Figure 6.7, where errors and warnings are displayed on the left and their resolution appears on the right.

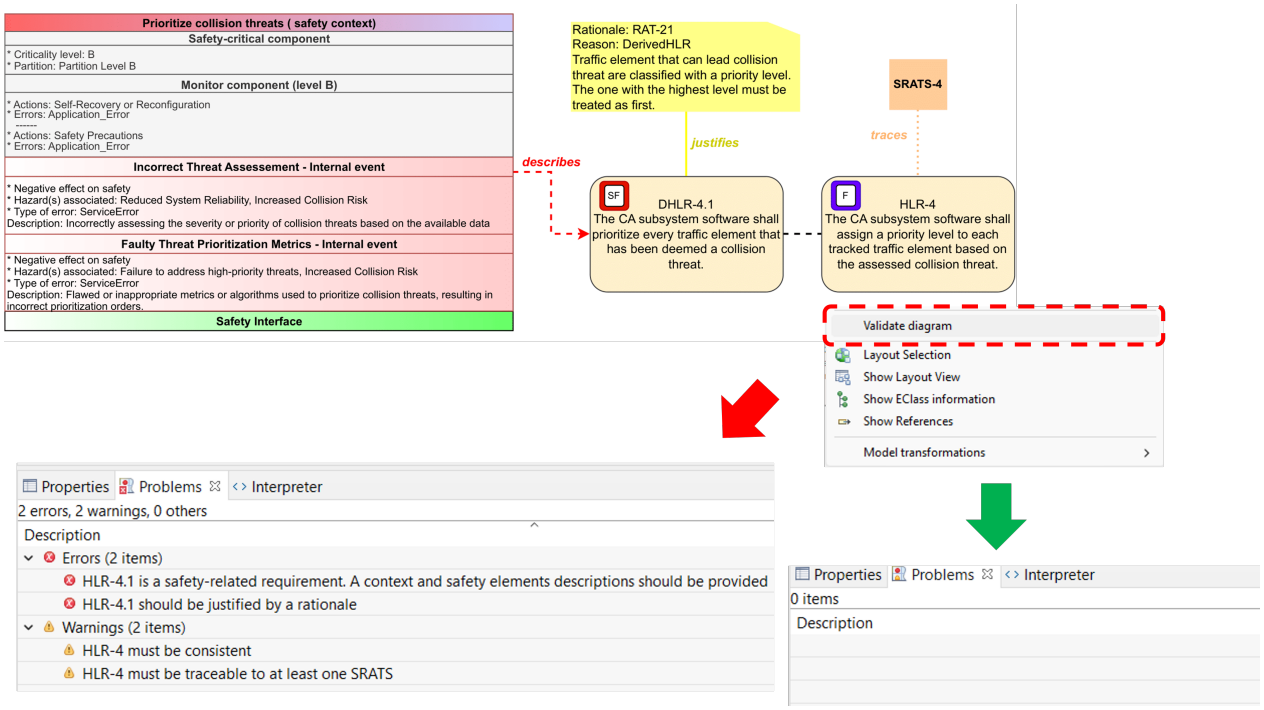

Figure 6.7 User interface for diagram validation in ReDaML. The figure displays HLR-4 and DHLR-4.1 models, as illustrated in Figure 5.4, alongside a context menu with the "Validate diagram" option. Validation results are displayed in the bottom left window for models with errors, while the bottom right window confirms an error-free model with no issues reported.

Once the requirements have been reviewed and the transition criteria between development phases have been met, the design process comprising architecture can be initialized. In our context, the architecture is generated using the proposed model transformation.

Activity 3 - Perform model transformation The transformation of the ReDaML model to the AADL architecture is performed straightforwardly within the Eclipse environment itself. Figure 6.8 shows the menu for generating AADL and the generated files that can be analyzed and refined in tools that support the architectural language, in our case using Osate[4].

As seen in Figure 6.8, five *aadl* files were generated. Each one contributes to the overall structure of the architecture. The *"DataTypes.aadl"* file is intended to define the data types used in the components. The *"ExternalComponents.aadl"* file generates software and hardware components that are integrated into the top-level system and connect directly with the main software.

[4]https://osate.org/

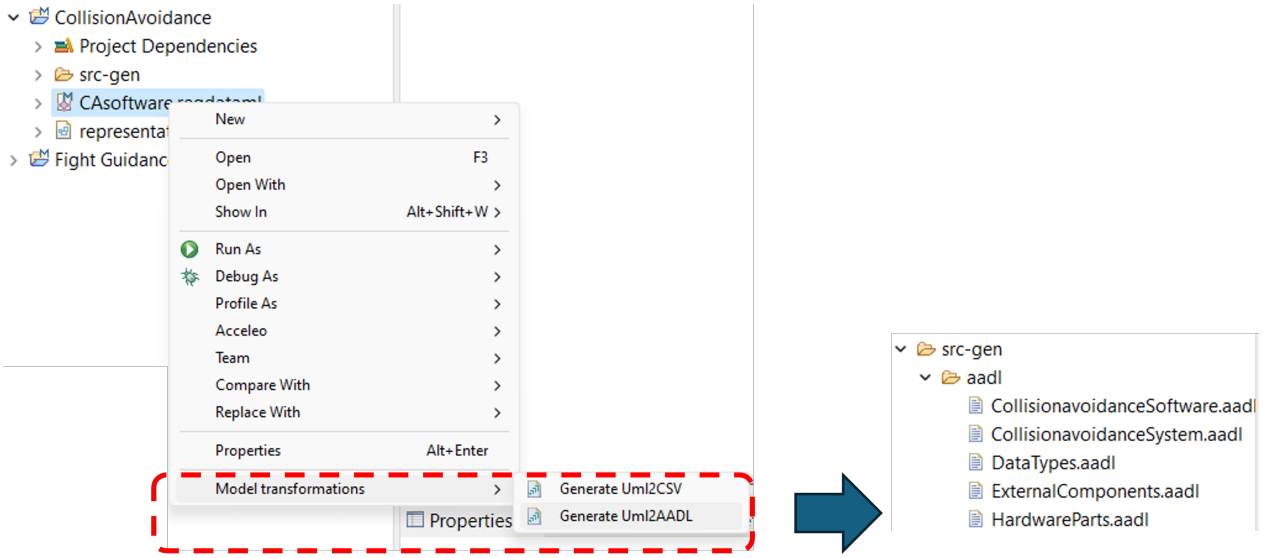

Figure 6.8 Transformation process in Eclipse environment. A menu is directly in the Eclipse tool where the HLR was modelled to facilitate the transformation. The image below represents the generated AADL files that form the architecture.

"HardwareParts.aadl" is a supplementary file that complements the architecture by specifying default resources that the application needs to run. It is also part of the ARINC653 template, which allows for the allocation and separation of space and time to each partition. This file contains resources such as a processor, a partition (represented by virtual processors in AADL), memory, and communication channels such as BUS. In addition, properties captured in the safety context, such as errors and potential actions configured in Health monitoring, are also specified.

The other two files correspond to the views of the architecture, in which one represents the main software, *"CollisionavoidanceSoftware.aadl"*, with its internal structure and relationship to the ARINC653 base, as well as safety-related details. The second file, *"CollisionavoidanceSystem.aadl"*, represents the view of the software architecture and the relationships with the other components.

The view of the system's architecture is shown in Figure 6.9. The outermost boundary represents the collision avoidance system, which is comprised of software components and external devices that work together to define the system's behaviour. In the center is the main software responsible for the functionality and logic presented at the beginning of the case study and the relationships with the other elements through the communication of data and events.

Figure 6.9 System view of architecture generated in AADL

6.4 Discussion & Evaluation

Throughout the mapping process, there were significant obstacles to overcome in the development of the software architecture-generating framework. It took a lot of collaboration and testing to smoothly transition from requirements data to a legitimate AADL architecture. To ensure that the models generated by the transformation process were free of syntactic and semantic issues, multiple iterations were needed. Our method's primary applicability in the avionics area stems from its dependence on an ARINC653 template architecture. This specificity guarantees a customized solution for safety-critical aviation systems, even though it may restrict its application across industries.

Despite these challenges, our framework has demonstrated several significant advantages. Firstly, it adheres to the standards outlined in DO-178C, ensuring compliance with regulatory requirements for safety-critical software development. DO-178C (annex A) objectives were successfully met through the structured processes implemented in our framework (Table 6.4). This adherence ensures not only regulatory compliance but also confidence in the reliability and safety of the resulting software architecture.

Moreover, the introduction of our framework resulted in a significant reduction in the time necessary for software architecture tasks. Comparative analysis with traditional manual methods experienced by our industrial partner revealed a staggering decrease of approximately 75% in the time expended on architecture development for the same quantity of HLR. The conventional approach in question entails creating an AADL architecture from scratch after carefully examining every requirement in natural language. The significant time-saving can be attributed to the automation and simplification of procedures enabled by our framework, demonstrating its effectiveness in increasing project efficiency and shortening development timelines.

Table 6.4 Summary of DO-178C objectives satisfied through the framework

Objective	Description	Details
A-2.3	Software architecture is developed.	Established by generating the AADL architecture using model transformation.
A-4.8	Software architecture is compatible with high-level requirements	Established through the model transformation. All requirements are taken into account by the transformation rule.
A-4.9	Software architecture is consistent.	Established through modeling using an architecture language with well-defined syntax and semantics, such as AADL.
A-4.10	Software architecture is compatible with target computer.	Established in accordance with the ARINC653 template, which allows details of the target computer to be included and verified using tools provided by the AADL environment.
A-4.11	Software architecture is verifiable.	Established through modelling using an architecture language with well-defined syntax and semantics, such as AADL. The AADL model can also be translated into formal languages to be verified.
A-4.12	Software architecture conforms to standards	Established through the model transformation that generates a valid architecture in AADL and takes into account internal standards in its transformation rules.
A-4.13	Software partitioning integrity is confirmed	Established in accordance with the ARINC653 template, which allows details of the target computer to be included and verified using tools provided by the AADL environment.

Furthermore, the adoption of AADL as the modelling language within our framework enables a plethora of critical analyses to be conducted. These assessments include schedulability, fault tolerance, and resource utilization, among others. The broad analytical capabilities provided by AADL enable developers to uncover potential system vulnerabilities and enhance performance, improving the overall quality and robustness of the software design. Comparative evaluations of architectures developed using our framework versus alternative approaches regularly emphasize the superior quality and adherence to safety-critical requirements obtained by our framework, reinforcing its importance in safety-critical software development contexts.

When compared to competing techniques, the use of our framework in software architecture within the DO-178C context has specific advantages. For example, when using the methodology described in [98], which includes a DSML for architecture, multiple iterative steps are frequently required to get the appropriate architecture. Additionally, engineers must undergo

extensive training to master the modelling language, leading to significant time and effort investments in architecture development. The need for tooling proficiency can also make it difficult to ensure the quality of the resulting architecture.

Similarly, the approach employed by [82], which uses a transformation to AADL architecture, has drawbacks. While focused on traceability, this technique encounters challenges with communication between components, particularly when dealing with data flows, potentially compromising the integrity and functionality of the architecture.

Furthermore, insights obtained from the study of deriving architectural models from requirements specifications, as demonstrated by [73], highlight the necessity of addressing both functional and non-functional requirements in safety-critical development. Many existing techniques prioritize functional features above key non-functional factors required for system safety and reliability. In contrast, our framework takes a more holistic approach, deliberately combining both functional and non-functional criteria into the resulting software architecture. By addressing this key feature, our approach improves the architecture's comprehensiveness and suitability for safety-critical applications, reducing possible risks and assuring regulatory compliance under the DO-178C framework.

6.5 Chapter Summary

In this chapter, we introduced a framework designed to automate the transition from requirements data to software architecture, addressing key challenges in safety-critical system development. By leveraging model transformation mechanisms, the framework ensures that both functional and non-functional requirements are systematically integrated into the architecture, maintaining traceability and compliance with DO-178C standards. This approach reduces the potential for errors, enhances the consistency between artifacts, and simplifies the development process, even for engineers with varying levels of expertise. The framework demonstrates how automation can streamline the creation of high-quality, reliable software architectures while adhering to the rigorous demands of safety-critical domains.

CHAPTER 7 CONCLUSION

7.1 Summary of Works

This thesis presents a cohesive contribution to the field of safety-critical software development, focusing on the challenges and opportunities in creating DO-178C-compliant software for UAVs. The research integrates three interrelated contributions to address the complexities of requirements specification, software architecture design, and certification: a comprehensive methodology combining formal methods and automated tools, a DSML for requirements specification, and a framework for converting HLRs into software architecture. These developments, when combined, create a comprehensive strategy for improving safety, efficiency, and compliance in aviation software development.

The proposed methodology combines the accuracy of formal methods with the effectiveness of automation, as shown by the LDRA Tool Suite, to ensure DO-178C compliance throughout the software lifecycle. Applying it to the development of a UAV collision avoidance system demonstrated several advantages, such as strict traceability, early inconsistency detection, adherence to the standard objectives, and evidence report generation. This approach minimizes the likelihood of defects and simplifies certification procedures by automating the creation of essential artifacts. Although obstacles like the steep learning curve for formal methods and reliance on tools were recognized, these were addressed through iterative improvements and an emphasis on real-world applicability.

An additional contribution is the design of ReDaML, a DSML that complies with DO-178C and is specifically intended to tackle the issues of HLR specification in safety-critical systems. ReDaML includes 13 informational needs derived from comprehensive research and offers a systematic, user-friendly method for gathering and validating safety-related aspects. ReDaML guarantees models' consistency and semantic correctness by integrating constraints with the *Object Constraint Language* (OCL). The language was confirmed via a case study conducted with an industrial partner, which underscored its ability to improve communication and analysis among multidisciplinary teams. Although ReDaML is based on aviation standards, its adaptability allows it to be used in various critical domains.

Building on ReDaML, the study introduced a framework for automatically converting HLRs into software architecture. This framework incorporates the DSML to define safety-related requirements and utilizes a transformation engine to create architecture in AADL according to established rules. By automating a procedure that typically depends on engineers'

knowledge and instincts, the framework streamlines the shift from requirements to architecture while ensuring compliance with DO-178C objectives. The case study illustrated how the framework can increase architectural design consistency, decrease manual effort, and streamline workflows.

The synergy among these contributions showcases a holistic method for tackling major issues in safety-critical software development. Collectively, they offer a methodical approach backed by stringent verification and validation procedures, from requirements definition to architectural design. This cohesive framework improves adherence to aviation regulations and promotes innovation by facilitating the effective creation of intricate systems. By mitigating the limitations of conventional methods, this study establishes itself as a significant step toward improving the state-of-the-art in safety-critical systems.

Regarding automation, while the proposed methodology integrates formal methods and model transformation to automate key aspects of the software development lifecycle, several activities remain manual by design. For instance, all peer reviews and analysis of formal verification results require human interpretation to ensure correctness in the context of domain-specific knowledge. Within ReDaML, the modelling of HLRs is performed manually by the engineer, although using a guided and structured approach that improves precision. In the architecture generation framework, although the model transformation process is automated, the validation of the generated architecture and its conformance to system-level constraints still relies on manual review. These manual components are crucial to align automation outputs with engineering judgment and certification expectations.

The methodology and supporting tools presented in this thesis were developed with DO-178C as the guiding standard. Consequently, several components—such as ReDaML's modelling structure and the verification objectives integrated into the workflow—reflect the terminology, assurance levels, and lifecycle activities defined by DO-178C. However, the core principles of the approach, namely formal requirement specification, model-based transformation, and early verification, are widely applicable across other safety-critical domains. Standards such as ISO 26262 (automotive), IEC 61508 (industrial systems), and EN 50128 (railway) also emphasize traceability, formal methods, and quality assurance. While terminology and certification objectives may differ, the structured modelling and transformation techniques can be adapted to these domains with appropriate tailoring.

The architecture pattern used in this work, however, is more domain-specific. The generated AADL architecture is based on ARINC 653, a standard for avionics software partitioning. Therefore, while the model transformation approach itself is extensible, the template model as implemented is tailored for avionics and would require adaptation for domains using different

architectural constraints or middleware standards.

While the case study and examples in this thesis focus on UAVs, most of the proposed contributions are not limited to drone platforms. The challenges addressed are common to a wide range of safety-critical embedded systems. The modularity and abstraction of the proposed methodology support reuse in other platforms, including manned aircraft and autonomous vehicles.

Aspects that are more specific to drones include assumptions about system autonomy, constrained hardware resources, and specific operational profiles typical of UAV missions, which are enforced in the project itself. Additionally, the focus on lightweight and cost-effective verification practices is aligned with the typical budget and certification constraints in the drone industry, which may differ from large-scale aerospace programs. Nonetheless, the core technical contributions remain applicable, provided that domain-specific adaptations are made for system constraints and regulatory frameworks.

In conclusion, this thesis's findings illustrate the benefits of integrating formal methods, domain-specific modelling, and automation to meet the rigorous standards of DO-178C. The proposed solutions address current regulatory requirements while anticipating future difficulties as UAV technologies progress. Applying this research effectively to real-world case studies creates a solid foundation for future progress in aviation software development, prioritizing innovation and safety within the industry.

7.2 Limitations

Although the proposed approach successfully combines formal methods and automated tools for DO-178C compliance, it has some limitations. A major issue is its restricted use during the project planning stage. Incorporating planning processes into the methodology would guarantee improved alignment between initial project plans and later development activities, enhancing overall compliance with certification goals. Moreover, integrating the methodology into the broader development lifecycle necessitates improvements to ensure that intended plans are reliably met. Another area for enhancement involves broadening the range of accessible tools and techniques, providing developers with more flexibility in overseeing traceability, automating tests, and augmenting formal methods for improved verification.

ReDaML, a specialized modeling language for defining high-level requirements, also has limitations. An important drawback is the graphical user interface, which becomes increasingly inefficient with rising software complexity, making the visualization and handling of complex models more difficult. Moreover, the manual procedure of modelling requirements that are

initially recorded in natural language creates inefficiencies and raises the likelihood of mistakes. The transition from natural language to structured models could be automated to improve usability and greatly lessen the cognitive demand on engineers.

The architecture generation framework, though effective in creating architectures that adhere to ARINC653, is limited by its dependence on set templates. This confines its use to different architectural standards or non-ARINC653 scenarios, reducing its adaptability for developers operating in various fields. Enlarging the framework to include diverse architectural models would increase its relevance and facilitate wider usage across sectors.

7.3 Future Research

The limitations presented above generate the following opportunities for future research:

begin

- **Enhancements to the Methodology:** begin
  - Integration into the project planning process to ensure alignment with initial plans throughout the software development lifecycle.
  - Development of complementary approaches to formal methods for increased flexibility and usability.
  - Ensuring seamless interoperability between different tools used in safety-critical software development.
- **Improvements to ReDaML:** begin
  - Refinement of the graphical user interface to handle complex models more effectively.
  - Addition of features for automated generation of certification evidence reports.
  - Development of mechanisms to automate input from requirements written in natural language.
- **Extension of the Architecture Generation Framework:** begin
  - Expanding support beyond ARINC653 to encompass other architectural standards.
  - Creation of a library of customizable templates for diverse operational contexts and certification requirements.

- Exploration of dynamic adaptation techniques for architecture generation rules to accommodate evolving system requirements.

REFERENCES

[1] N. G. Leveson, *Engineering a safer world: Systems thinking applied to safety.* The MIT Press, 2016.

[2] D. Wanner, H. A. Hashim, S. Srivastava, and A. Steinhauer, "Uav avionics safety, certification, accidents, redundancy, integrity, and reliability: a comprehensive review and future trends," *Drone Systems and Applications*, vol. 12, pp. 1–23, 2024.

[3] R. Inc, "Do-178c - software considerations in airborne systems and equipment certification," RTCA, Inc, Dec. 2011.

[4] S. B. Nazarudeen and J. Liscouët, "State-of-the-art and directions for the conceptual design of safety-critical unmanned and autonomous aerial vehicles," in *2021 IEEE International Conference on Autonomous Systems (ICAS).* IEEE, 2021, pp. 1–5.

[5] W. K. Youn, S. B. Hong, K. R. Oh, and O. S. Ahn, "Software certification of safety-critical avionic systems: Do-178c and its impacts," *IEEE Aerospace and Electronic Systems Magazine*, vol. 30, no. 4, pp. 4–13, 2015.

[6] J. E. F. Ribeiro, J. G. Silva, and A. Aguiar, "Weaving agility in safety-critical software development for aerospace: From concerns to opportunities," *IEEE Access*, vol. 12, pp. 52 778–52 802, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:269106376

[7] L. Rierson, *Developing safety-critical software: a practical guide for aviation software and DO-178C compliance.* CRC Press, 2017.

[8] K. Dmitriev, S. A. Zafar, K. Schmiechen, Y. Lai, M. Saleab, P. Nagarajan, D. Dollinger, M. Hochstrasser, F. Holzapfel, and S. Myschik, "A lean and highly-automated model-based software development process based on do-178c/do-331," in *2020 AIAA/IEEE 39th Digital Avionics Systems Conference (DASC).* IEEE, 2020, pp. 1–10.

[9] J. Pyrgies, D. Gigan, and R. Haelterman, "An innovative approach for achieving do-178c certification of an intelligent system implementing sense-and-avoid function in uavs," in *Air Transport Research Society World Conference*, 2017.

[10] N. B. Ruparelia, "Software development lifecycle models," *ACM SIGSOFT Software Engineering Notes*, vol. 35, no. 3, pp. 8–13, 2010.

[11] D. W. Loveland, *Automated theorem proving: A logical basis.* Elsevier, 2016.

[12] C. Baier and J.-P. Katoen, *Principles of model checking.* MIT press, 2008.

[13] D. L. Lempia and S. P. Miller, "Requirements engineering management handbook," *National technical information service (ntis)*, vol. 1, 2009.

[14] A. E. E. Committee *et al.*, "Avionics application software standard interface," *Arinc Specifcation*, vol. 653, 1997.

[15] M. H. ter Beek, R. Chapman, R. Cleaveland, H. Garavel, R. Gu, I. ter Horst, J. J. Keiren, T. Lecomte, M. Leuschel, K. Y. Rozier *et al.*, "Formal methods in industry," *Formal Aspects of Computing*, 2024.

[16] R. F. SC-205, *Formal Methods Supplement to DO-178C and DO-278A.* RTCA, Incorporated, 2011.

[17] G. Gigante and D. Pascarella, "Formal methods in avionic software certification: the do-178c perspective," in *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation.* Springer, 2012, pp. 205–215.

[18] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *Future of Software Engineering (FOSE'07).* IEEE, 2007, pp. 37–54.

[19] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson, "Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice," *Software & Systems Modeling*, vol. 17, pp. 91–113, 2018.

[20] T. Clark and B. Barn, "Domain engineering for software tools," *Domain Engineering: Product Lines, Languages, and Conceptual Models*, 2013.

[21] D. Pilone and N. Pitman, *UML 2.0 in a Nutshell.* " O'Reilly Media, Inc.", 2005.

[22] J. Gray, S. Neema, J.-P. Tolvanen, A. S. Gokhale, S. Kelly, and J. Sprinkle, "Domain-specific modeling." *Handbook of dynamic system modeling*, vol. 7, pp. 7–1, 2007.

[23] J. E. Rivera, "On the semantics of real-time domain specific modeling languages," Ph.D. Thesis, Universidad de Málaga, Departamento de Lenguajes y Ciencias de la Computación, Málaga, Spain, 2015.

[24] J. E. Rivera, J. R. Romero, and A. Vallecillo, "Behavior, time and viewpoint consistency: Three challenges for mde," in *International Conference on Model Driven Engineering Languages and Systems.* Springer, 2008, pp. 60–65.

[25] S. Robert, S. Gérard, F. Terrier, and F. Lagarde, "A lightweight approach for domain-specific modeling languages design," in *2009 35th Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, 2009, pp. 155–161.

[26] K. Czarnecki, S. Helsen *et al.*, "Classification of model transformation approaches," in *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, vol. 45, no. 3. USA, 2003, pp. 1–17.

[27] A. G. Kleppe, J. B. Warmer, and W. Bast, *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003.

[28] T. Mens and P. Van Gorp, "A taxonomy of model transformation," *Electronic notes in theoretical computer science*, vol. 152, pp. 125–142, 2006.

[29] P. H. Feiler, D. P. Gluch, and J. J. Hudak, "The architecture analysis & design language (aadl): An introduction," Software Engineering Institute, Carnegie Mellon University, Tech. Rep., 2006.

[30] P. H. Feiler and D. P. Gluch, *Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language*. Addison-Wesley, 2012.

[31] Y. Zhao and D. Ma, "Embedded real-time system modeling and analysis using aadl," in *2010 International Conference on Networking and Information Technology*. IEEE, 2010, pp. 247–251.

[32] J. Dick, E. Hull, K. Jackson, J. Dick, E. Hull, and K. Jackson, "Doors: A tool to manage requirements," *Requirements Engineering*, pp. 187–206, 2017.

[33] *TBmanager User Guide*, LDRA, 2022.

[34] May 2024. [Online]. Available: https://ldra.com/products/ldra-tool-suite/

[35] *TBvision User Guide*, LDRA, 2022.

[36] *TBrun Tutorial*, LDRA, 2022.

[37] D. Jackson, *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[38] G. J. Holzmann, "The model checker spin," *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[39] M. R. Gadelha, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, and D. A. Nicole, "Esbmc 5.0: an industrial-strength c model checker," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 888–891.

[40] C. Barrett and C. Tinelli, "Satisfiability modulo theories," *Handbook of model checking*, pp. 305–343, 2018.

[41] W. W. Royce, "Managing the development of large software systems: concepts and techniques," in *Proceedings of the 9th international conference on Software Engineering*, 1987, pp. 328–338.

[42] INCOSE, *INCOSE systems engineering handbook*. John Wiley & Sons, 2023.

[43] P. Zakrzewski, J. Narkiewicz, and D. Brenchley, "Safety critical software development methodologies in avionics," *Transactions on Aerospace Research*, vol. 2020, no. 2, pp. 59–71, 2020.

[44] S. Basagiannis, "Software certification of airborne cyber-physical systems under do-178c," in *2016 International Workshop on symbolic and numerical methods for reachability analysis (SNR)*. IEEE, 2016, pp. 1–6.

[45] N. Metayer, A. Paz, and G. El Boussaidi, "Modelling do-178c assurance needs: A design assurance level-sensitive dsl," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2019, pp. 338–345.

[46] A. Paz and G. E. Boussaidi, "Building a software requirements specification and design for an avionics system: An experience report," in *33rd Annual ACM Symposium on Applied Computing*, 2018.

[47] P. Panchal, D. Surmann, and S. Myschik, "Further development of a flight control software for a lift-to-cruise vehicle," in *AIAA SCITECH 2025 Forum*, 2025, p. 2327.

[48] J. Pyrgies, "Towards do-178c certification of adaptive learning uav agents designed with a cognitive architecture," in *Proceedings of the ACM/IEEE 42nd international conference on software engineering: Companion proceedings*, 2020, pp. 174–177.

[49] D. Brown, H. Delseny, K. Hayhurst, and V. Wiels, "Guidance for using formal methods in a certification context," in *Embedded Real Time Software and Systems Conference*, no. NF1676L-10457, 2010.

[50] M. Elqortobi, W. El-Khouly, A. Rahj, J. Bentahar, and R. Dssouli, "Verification and testing of safety-critical airborne systems: a model-based methodology," *Computer Science and Information Systems*, vol. 17, no. 1, pp. 271–292, 2020.

[51] D. Bhatt, B. Hall, A. Murugesan, D. Oglesby, E. Bush, E. Engstrom, J. Mueller, and M. Pelican, "Opportunities and challenges for formal methods tools in the certification of avionics software," in *2017 IEEE Aerospace Conference*. IEEE, 2017, pp. 1–20.

[52] D. Cofer and S. Miller, "Do-333 certification case studies," in *NASA Formal Methods: 6th International Symposium, NFM 2014, Houston, TX, USA, April 29–May 1, 2014. Proceedings 6*. Springer, 2014, pp. 1–15.

[53] D. Cofer and S. P. Miller, "Formal methods case studies for do-333," Tech. Rep., 2014.

[54] L. G. Wagner, D. Cofer, K. Slind, C. Tinelli, and A. Mebsout, "Formal methods tool qualification," NASA, Hampton, VA, Tech. Rep. NASA/CR-2017-219371, 2017. [Online]. Available: https://shemesh.larc.nasa.gov/fm/FMinCert/NASA-CR-2017-219371.pdf

[55] M. Webster, M. Fisher, N. Cameron, and M. Jump, "Formal methods for the certification of autonomous unmanned aircraft systems," in *Computer Safety, Reliability, and Security: 30th International Conference, SAFECOMP 2011, Naples, Italy, September 19-22, 2011. Proceedings 30*. Springer, 2011, pp. 228–242.

[56] J. C. Santos, A. Shokri, and M. Mirakhorli, "Towards automated evidence generation for rapid and continuous software certification," in *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2020, pp. 287–294.

[57] P. Panchal, K. Dmitriev, S. Myschik, and F. Holzapfel, "Comprehensive overview of a process-oriented build tool for airborne safety-critical software development," in *2023 10th International Conference on Recent Advances in Air and Space Technologies (RAST)*. IEEE, 2023, pp. 01–06.

[58] L. Bao, C. Fuhrman, and R. Landry, "Certification considerations of software-defined radio using model-based development and automated testing," in *2023 IEEE/AIAA 42nd Digital Avionics Systems Conference (DASC)*. IEEE, 2023, pp. 1–8.

[59] M. Webster, N. Cameron, M. Fisher, and M. Jump, "Generating certification evidence for autonomous unmanned aircraft using model checking and simulation," *Journal of Aerospace Information Systems*, vol. 11, no. 5, pp. 258–279, 2014.

[60] A. Khasanov, V. Shishkin, and K. Larin, "Automation of software avionics verification in accordance with do-178c standard," in *2022 VIII International Conference on Information Technology and Nanotechnology (ITNT)*. IEEE, 2022, pp. 1–5.

[61] S. Paul, C. Alexander, M. Durling, K. Siu, D. Prince, B. Meng, S. C. Varanasi, and D. Stuart, "Automated do-178c compliance summary through evidence curation," in *2023 IEEE/AIAA 42nd Digital Avionics Systems Conference (DASC)*. IEEE, 2023, pp. 1–10.

[62] J.-P. BODEVEIX, "Modeling languages for requirements engineering and quantitative analysis of embedded systems," Ph.D. dissertation, Université de Bretagne-Sud, 2013.

[63] L. E. G. Martins and T. Gorschek, "Requirements engineering for safety-critical systems: A systematic literature review," *Information and software technology*, 2016.

[64] B. Nuseibeh and S. Easterbrook, "Requirements engineering: a roadmap," in *Proceedings of the Conference on the Future of Software Engineering*, 2000.

[65] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese, "Requirements specification for process-control systems," *IEEE transactions on software engineering*, vol. 20, no. 9, pp. 684–707, 1994.

[66] M. W. Whalen, "A formal semantics for the requirements state machine language without events," Ph.D. dissertation, UNIVERSITY OF MINNESOTA.

[67] A. W. Fifarek, L. G. Wagner, J. A. Hoffman, B. D. Rodes, M. A. Aiello, and J. A. Davis, "Spear v2. 0: Formalized past ltl specification and analysis of requirements," in *NASA Formal Methods: 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings 9*. Springer, 2017, pp. 420–426.

[68] A. Paz and G. El Boussaidi, "A requirements modelling language to facilitate avionics software verification and certification," in *IEEE/ACM 6th International Workshop on Requirements Engineering and Testing*. IEEE, 2019.

[69] G. Zoughbi, L. Briand, and Y. Labiche, "A uml profile for developing airworthiness-compliant (rtca do-178b), safety-critical software," in *Model Driven Engineering Languages and Systems: 10th International Conference, MoDELS 2007, Nashville, USA, September 30-October 5, 2007. Proceedings 10*. Springer, 2007, pp. 574–588.

[70] ——, "Modeling safety and airworthiness (rtca do-178b) information: conceptual model and uml profile," *Software & Systems Modeling*, vol. 10, pp. 337–367, 2011.

[71] P. Micouin, "Toward a property based requirements theory: System requirements structured as a semilattice," *Systems engineering*, 2008.

[72] A. Alebrahim and M. Heisel, *Bridging the gap between requirements engineering and software architecture*. Springer, 2017.

[73] E. Souza, A. Moreira, and M. Goulão, "Deriving architectural models from requirements specifications: A systematic mapping study," *Information and software technology*, vol. 109, pp. 26–39, 2019.

[74] T. Eisenreich, S. Speth, and S. Wagner, "From requirements to architecture: an ai-based journey to semi-automatically generate software architectures," in *Proceedings of the 1st International Workshop on Designing Software*, 2024, pp. 52–55.

[75] O. T. Erlich and D. H. Lorenz, "Optimal software architecture from initial requirements: An end-to-end approach," *arXiv preprint arXiv:2012.15533*, 2020.

[76] B. I. Ya'u and M. N. Yusuf, "Building software component architecture directly from user requirements," *International Journal of Engineering and Computer Science*, vol. 7, no. 2, pp. 23 557–23 566, 2018.

[77] H. Kaindl, "Model-based transition from requirements to high-level software design," in *Product-Focused Software Process Improvement: 14th International Conference, PROFES 2013, Paphos, Cyprus, June 12-14, 2013. Proceedings 14*. Springer, 2013, pp. 367–369.

[78] Z. Durdik, "Towards a process for architectural modelling in agile software development," in *Proceedings of the joint ACM SIGSOFT conference–QoSA and ACM SIGSOFT symposium–ISARCS on Quality of software architectures–QoSA and architecting critical systems–ISARCS*, 2011, pp. 183–192.

[79] G. Loniewski, A. Armesto, and E. Insfran, "An architecture-oriented model-driven requirements engineering approach," in *2011 Model-Driven Requirements Engineering Workshop*. IEEE, 2011, pp. 31–38.

[80] R. F. Passarini, J.-M. Farines, J. M. Fernandes, and L. B. Becker, "Cyber-physical systems design: transition from functional to architectural models," *Design Automation for Embedded Systems*, vol. 19, pp. 345–366, 2015.

[81] L. Chung, S. Supakkul, N. Subramanian, J. L. Garrido, M. Noguera, M. V. Hurtado, M. L. Rodríguez, and K. Benghazi, "Goal-oriented software architecting," *Relating software requirements and architectures*, pp. 91–109, 2011.

[82] F. Wang, Z.-B. Yang, Z.-Q. Huang, C.-W. Liu, Y. Zhou, J.-P. Bodeveix, and M. Filali, "An approach to generate the traceability between restricted natural language requirements and aadl models," *IEEE Transactions on Reliability*, vol. 69, no. 1, pp. 154–173, 2019.

[83] NASA ACCESS 5, "Collision avoidance functional requirements for step 1," NASA Dryden Flight Research Center, Tech. Rep., February 2006, revision 6.

[84] P. H. Feiler, B. Lewis, S. Vestal, and E. Colbert, "An overview of the sae architecture analysis & design language (aadl) standard: A basis for model-based architecture-driven embedded systems engineering," in *IFIP World Computer Congress, TC 2*. Springer, 2004, pp. 3–15.

[85] P. Feiler, "Open source aadl tool environment (osate)," in *AADL Workshop, paris*, 2004, pp. 1–40.

[86] Aeronautical Radio, Incorporated (ARINC), *ARINC 653: Avionics Application Software Standard Interface*, Std., 1997, prepared by the Airlines Electronic Engineering Committee (AEEC).

[87] *MATLAB Simulink User's Guide*, The MathWorks, Inc., Natick, Massachusetts, United States, 2023, https://www.mathworks.com/products/simulink.html.

[88] *MISRA C:2012 Guidelines for the use of the C language in critical systems*, MISRA (Motor Industry Software Reliability Association), 2013, iSBN: 978-1-906400-10-1, Available at: https://www.misra.org.uk/.

[89] A. Biere, "Bounded model checking," in *Handbook of satisfiability*. IOS press, 2021, pp. 739–764.

[90] U. Frank, "Domain-specific modeling languages: requirements analysis and design guidelines," *Domain engineering: Product lines, languages, and conceptual models*, 2013.

[91] H. A. Misson, R. Zrelli, M. Ben Attia, F. G. Magalhaes, and G. Nicolescu, "Redaml: A modeling language for do-178c high-level requirements in airspace systems," in *Proceedings of the 34th International Workshop on Rapid System Prototyping*, 2023, pp. 1–7.

[92] S. ARP4761, "Guidelines and methods for conducting the safety assessment process on airborne systems and equipments," *USA: The Engineering Society for Advancing Mobility Land Sea Air and Space*, 1996.

[93] J. B. Warmer and A. G. Kleppe, *The object constraint language: getting your models ready for MDA*. Addison-Wesley Professional, 2003.

[94] N. C. Team, "Collision avoidance functional requirements for step 1," *NASA access*, 2006.

[95] J. Stoker and A. Simpson, "Functional hazard assessment/preliminary system safety assessment (fha/pssa) report for military uav as oat outside segregated airspace," Tech. Rep.

[96] OMG, *OMG Meta Object Facility (MOF) Core Specification, Version 2.5.1*, Object Management Group Publication, Rev. 2.5.1, 2019. [Online]. Available: https://www.omg.org/spec/MOF/2.5.1/

[97] D. Firesmith, "Engineering safety requirements, safety constraints, and safety-critical requirements," *Journal of Object technology*, vol. 3, no. 3, pp. 27–42, 2004.

[98] J. Wu, T. Yue, S. Ali, and H. Zhang, "A modeling methodology to facilitate safety-oriented architecture design of industrial avionics software," *Software: Practice and experience*, vol. 45, no. 7, pp. 893–924, 2015.

APPENDIX A COLLISION AVOIDANCE SOFTWARE DEVELOPMENT - REQUIREMENTS AND DESIGN

The appendix provides a comprehensive collection of artifacts developed during the case study, illustrating the practical application of the proposed methodologies and frameworks for software development of the UAV Collision Avoidance System (CAS).

The following SRATS corresponds to the CAS system requirements that must be respected by the software, which will be verified later through test procedures.

System Requirements Allocated to Software (SRATS):

SRATS_001 *Detect Traffic*: The Collision Avoidance System shall detect traffic within its surveillance volume.
Rationale: Effective traffic detection is the first step in collision avoidance, ensuring that potential threats are identified early.
Category: Functional
Traceability: Developed into HLR_001

SRATS_002 *Detect Traffic*: The Collision Avoidance System shall calculate its surveillance volume depending on all the following conditions:

- detection range;
- azimuth field of regard;
- elevation field of regard.

Rationale: Defines the area within which traffic detection is performed.
Category: Functional
Traceability: Developed into HLR_001

SRATS_003 *Detect Traffic*: The Collision Avoidance System shall detect cooperative traffic at a range of at least 20 nautical miles.
Rationale: Ensures the system can detect cooperative traffic, which is essential for collision avoidance.
Category: Functional
Traceability: Developed into HLR_002

SRATS_004 *Detect Traffic*: The Collision Avoidance System shall detect cooperative traffic within an azimuth FOR of at least +/-110° referenced from the flight path of the UA.

Rationale: Ensures wide horizontal coverage for detecting traffic.

Category: Functional

Traceability: Developed into HLR_003

SRATS_005 *Detect Traffic*: The Collision Avoidance System shall detect cooperative traffic within an elevation FOR of at least +/-15° referenced from the flight path of the UA.

Rationale: Ensures vertical coverage for detecting traffic.

Category: Functional

Traceability: Developed into HLR_003

SRATS_006 *Detect Traffic*: The average Collision Avoidance System detection rate shall be equal to or greater than 1.0 hertz.

Rationale:A sufficient detection rate is necessary to ensure timely updates and responses to detected traffic.

Category: Functional

Traceability: Developed into HLR_004

SRATS_007 *Track Traffic*: The Collision Avoidance System shall track the detected traffic.

Rationale: Accurate tracking of traffic allows for continuous monitoring and assessment of potential collision threats.

Category: Functional

Traceability: Developed into HLR_005

SRATS_008 *Track Traffic*: The Collision Avoidance System shall track cooperative traffic within an azimuth FOR of at least +/-110° and elevation FOR of at least +/-15° referenced from the flight path of the UA.

Category: Functional

Traceability: Developed into HLR_005

SRATS_009 *Evaluate Collision Potential*: The Collision Avoidance System shall evaluate the potential for collision with each tracked traffic element, including the assessment of existing collision threats.

Rationale: Evaluating collision potential helps in identifying imminent threats and preparing the system to take appropriate actions.

Category: Functional

Traceability: Developed into HLR_006

SRATS_010 *Evaluate Collision Potential*: The Collision Avoidance System shall continuously determine if any detected traffic elements pose a collision threat to the vehicle.
Category: Functional
Traceability: Developed into HLR_006

SRATS_011 *Prioritize Collision Threats*: The Collision Avoidance System shall prioritize the traffic posing a collision threat.
Rationale: Prioritizing threats ensures that the most immediate and dangerous threats are addressed first, optimizing response time and effectiveness.
Category: Functional
Traceability: Developed into HLR_007

SRATS_012 *Determine Avoidance Maneuver*: The Collision Avoidance System shall determine an avoidance maneuver that prevents a collision.
Rationale: Determining the correct avoidance maneuver is essential to ensure the safety of the UAV and other airspace users.
Category: Functional
Traceability: Developed into HLR_008

SRATS_013 *Determine Avoidance Maneuver*: The Collision Avoidance System shall revise the maneuver recommendation when other aircraft are simultaneously maneuvering.
Category: Functional
Traceability: Developed into HLR_008

SRATS_014 *Command Maneuver*: The Collision Avoidance System shall command an appropriate avoidance maneuver.
Rationale: Executing the correct maneuver ensures the UAV can avoid collisions effectively.
Category: Functional
Traceability: Developed into HLR_009

SRATS_015 *Sensor Data Integrity*: The Collision Avoidance System shall verify the integrity and accuracy of the sensor data before using it for detection and tracking.
Rationale: Ensuring that the sensor data is reliable and accurate is critical for all subsequent functions of the Collision Avoidance System.
Category: Performance
Traceability: Developed into HLR_010

SRATS_016 *System Self-Test*: The Collision Avoidance System shall perform self-tests on startup and periodically during operation to ensure all components are functioning correctly.
Rationale: Regular self-tests help in identifying and mitigating failures that could compromise the system's ability to detect and avoid collisions.
Category: Safety
Traceability: Developed into HLR_011

SRATS_017 *Redundancy Management*: The Collision Avoidance System shall have redundancy management capabilities to handle failures in primary detection and tracking components.
Rationale: Redundancy ensures that the system remains operational even if some components fail, thus maintaining safety.
Category: Safety
Traceability: Developed into HLR_012

SRATS_018 *Communication with Ground Control*: The Collision Avoidance System shall communicate status and alerts to the ground control station in real-time.
Rationale: Real-time communication with ground control allows for human intervention when necessary and provides situational awareness to operators.
Category: Non-Functional
Traceability: Developed into HLR_013

SRATS_019 *False Alarm Management*: The Collision Avoidance System shall minimize false alarms to prevent unnecessary maneuvers.
Rationale: Reducing false alarms ensures that the system does not execute unnecessary maneuvers, which could be disruptive or hazardous.
Category: Performance
Traceability: Developed into HLR_014

The SRATS is one of the inputs of the Requirements process, which aims to define the software requirements at a higher level. As the HLRs correspond to the software level, more details are introduced compared with system ones, such as the more detailed interface between computable assets and more specific data coming from external sources. Requirements Data is the set of HLRs and possible DHLRs developed in this phase.

High-Level Requiements:

HLR_001 *Traffic Detection*: The Collision Avoidance System shall detect traffic within its surveillance volume. Note: The surveillance volume is defined by three performance characteristics of the sensor: detection range, azimuth field of regard, and elevation field of regard.
Rationale: Ensuring that traffic within the defined surveillance volume is detected is critical for initiating subsequent collision avoidance processes.
Traceability:
Developed from SRATS_001 and SRATS_002.
Software Allocation: DETECT_TRAFFIC
Developed into LLR-001, LLR-002, LLR-003, LLR-004, LLR-005, LLR-006, LLR-007, LLR-008, and LLR-009.

HLR_002 *Traffic Detection*: The Collision Avoidance System shall detect cooperative traffic at a range of at least 20 nautical miles.
Rationale: Ensuring the detection of cooperative traffic within a significant range is essential for collision avoidance.
Traceability:
Developed from SRATS_003.
Software Allocation: DETECT_TRAFFIC
Developed into LLR-0010.

HLR_003 *Traffic Detection*: The Collision Avoidance System shall detect cooperative traffic within an azimuth field of regard of at least $\pm 110°$ and an elevation field of regard of at least $\pm 15°$, referenced from the flight path of the UAV.
Rationale: Detecting traffic within these fields of regard ensures comprehensive surveillance around the UAV.
Traceability:
Developed from SRATS_004 and SRATS_005.
Software Allocation: DETECT_TRAFFIC
Developed into LLR-0011 and LLR-0012.

HLR_004 *Traffic Detection*: The average Collision Avoidance System detection rate shall be equal to or greater than 1.0 hertz.
Rationale: A sufficient detection rate is necessary to ensure timely updates and responses to detected traffic.
Traceability:
Developed from SRATS_006.
Software Allocation: DETECT_TRAFFIC
Developed into LLR-0013.

HLR_005 *Traffic Tracking*: The Collision Avoidance System shall track the detected traffic. Note: The track is established when a state estimate is developed with sufficient confidence, and this estimate includes the traffic element's position and velocity vector.

Rationale: Accurate tracking of detected traffic is essential for evaluating potential collision threats and determining appropriate avoidance maneuvers.

Traceability:

Developed from SRATS_007 and SRATS_008.

Software Allocation: DETECT_TRAFFIC

Developed into LLR-0014, LLR-0015, LLR-0016, and LLR-0017.

HLR_006 *Collision Evaluation*: The Collision Avoidance System shall evaluate the potential for collision with each traffic element being tracked. This evaluation includes assessing existing collision threats.

Rationale: Evaluating collision potential is necessary to identify imminent threats and prepare the system for appropriate responses.

Traceability:

Developed from SRATS_009 and SRATS_010.

Software Allocation: DETECT_TRAFFIC

Developed into LLR-0018 and LLR-0019.

HLR_007 *Threat Prioritization*: The Collision Avoidance System shall prioritize the traffic posing a collision threat. Prioritization is based on ranking the time to collision of the identified threats.

Rationale: Prioritizing threats ensures that the most immediate and dangerous threats are addressed first, optimizing the system's response time and effectiveness.

Traceability:

Developed from SRATS_011.

Software Allocation: DETECT_TRAFFIC

Developed into LLR-0020 and LLR-0021.

HLR_008 *Maneuver Determination*: The Collision Avoidance System shall autonomously determine an avoidance maneuver that prevents a collision.
Rationale: Autonomous determination of avoidance maneuvers is critical to ensure timely and effective responses to imminent collision threats.
Traceability:
Developed from SRATS_012 and SRATS_013.
Software Allocation: DETECT_TRAFFIC
Developed into LLR-0022 and LLR-0023.

HLR_009 *Maneuver Command*: The Collision Avoidance System shall command an appropriate avoidance maneuver. Note: The commanded maneuver can include initiating a new maneuver, continuing an ongoing maneuver, or terminating an avoidance maneuver if a collision threat no longer exists.
Rationale: Executing the appropriate maneuver ensures the avoidance of collisions based on real-time evaluations of traffic and threats.
Traceability:
Developed from SRATS_014.
Software Allocation: DETECT_TRAFFIC into LLR-0024, LLR-0025, and LLR-0026.

HLR_010 *Sensor Data Integrity*: The CAS shall verify the integrity and accuracy of the sensor data before using it for detection and tracking.
Rationale: Ensuring that the sensor data is reliable and accurate is critical for all subsequent functions of the CAS.
Traceability:
Developed from SRATS_015.
Software Allocation: DETECT_TRAFFIC
Developed into LLR-0027.

HLR_011 *System Self-Test*: The CAS shall perform self-tests on startup and periodically during operation to ensure all components are functioning correctly.
Rationale: Regular self-tests help in identifying and mitigating failures that could compromise the system's ability to detect and avoid collisions.
Traceability:
Developed from SRATS_016.
Software Allocation: DETECT_TRAFFIC
Developed into LLR-0028 and LLR-0029.

HLR_012 *Redundancy Management*: The CAS shall have redundancy management capabilities to handle failures in primary detection and tracking components.
Rationale: Redundancy ensures that the system remains operational even if some components fail, thus maintaining safety.
Traceability:
Developed from SRATS_017.
Software Allocation: DETECT_TRAFFIC
Developed into LLR-0030.

HLR_013 *Communication with Ground Control*: The CAS shall communicate status and alerts to the ground control station in real-time.
Rationale: Real-time communication with ground control allows for human intervention when necessary and provides situational awareness to operators.
Traceability:
Developed from SRATS_018.
Software Allocation: DETECT_TRAFFIC
Developed into LLR-0031

HLR_014 *False Alarm Management*: The CAS shall minimize false alarms to prevent unnecessary maneuvers.
Rationale: Reducing false alarms ensures that the system does not execute unnecessary maneuvers, which could be disruptive or hazardous.
Traceability:
Developed from SRATS_019.
Software Allocation: DETECT_TRAFFIC
Developed into LLR-0032.

Design Description consists of the output artifacts created in the Design process. It includes the LLR and possible DLLR and the software architecture. The low-level requirements are written at the implementation level, where they are used to code the software. The architecture is important as it exposes the structure and relationship view of the software with the other components, which guides the developers in the source code development.

Low-Level Requirements:

LLR_001 *Sensor Status Verification and SensorInput Retrieval*: When the SensorStatus is True, the TrafficDetection shall retrieve the SensorInput. Otherwise, the TrafficDetection shall call the Alert function.
Traceability: Developed from HLR-001.

LLR_002 *SensorInput Data Validation*: When SensorInput is received, TrafficDetection shall validate the data as follows: - DetectionRange value is between 0.2 and 3000 meters. -AzimuthFOR value is between -110 and 110 degrees. - ElevationFOR value is between -15 and 15 degrees.
Traceability: Developed from HLR-001.

LLR_003 *Get Orientation*: When SensorInput data is validated, TrafficDetection shall get the UAV orientation using the OrientationData from SensorInput.
Traceability: Developed from HLR-001.

LLR_004 *Get Position*: When SensorInput data is validated, TrafficDetection shall get the current position using the PositionData from SensorInput.
Traceability: Developed from HLR-001.

LLR_005 *Get Sensor Measures*: When SensorInput data is validated, TrafficDetection shall get the UAV sensor measures using the SensorMeasures from SensorInput.
Traceability: Developed from HLR-001.

LLR_006 *Extract Traffic Position*: When the SensorMeasure is not empty, TrafficDetection shall calculate the TrafficPosition using the SensorMeasure and the CurrentPosition.
Traceability: Developed from HLR-001.

LLR_007 *Conflict Region Calculation*: TrafficDetection shall calculate the ConflictRegion using the CurrentPosition, HorizontalRadius, and VerticalMeasure.
Traceability: Developed from HLR-001.

LLR_008 *Verify Traffic Position Against Conflict Region*: TrafficDetection shall compare the TrafficPosition with the ConflictRegion and do the following: When TrafficPosition is in the ConflictRegion, set PotentialTraffic to True. Otherwise, set PotentialTraffic to False.
Traceability: Developed from HLR-001.

LLR_009 *Output Traffic Detection*: When PotentialTraffic is True, TrafficDetection shall output TrafficDetected, TrafficPosition, and CurrentPosition.
Traceability: Developed from HLR-001.

LLR_010 *Detection Range Validation*: TrafficDetection shall validate that the detection range for cooperative traffic is at least 20 nautical miles.
Traceability: Developed from HLR-002.

LLR_011 *Azimuth FOR Validation*: TrafficDetection shall validate that the azimuth field of regard for cooperative traffic detection is at least ±110°.
Traceability: Developed from HLR-003.

LLR_012 *Elevation FOR Validation*: TrafficDetection shall validate that the elevation field of regard for cooperative traffic detection is at least ±15°.
Traceability: Developed from HLR-003.

LLR_013 *Detection Rate Validation*: TrafficDetection shall ensure that the detection rate is equal to or greater than 1.0 hertz.
Traceability: Developed from HLR-004.

LLR_014 *Establish Track File*: When TrafficDetected is True, the TrafficTracking shall establish a track file for the detected traffic, including Position and VelocityVector.
Traceability: Developed from HLR-005.

LLR_015 *Update Track File*: The TrafficTracking shall update the track file for each detected traffic element at a rate of at least X updates per second.
Traceability: Developed from HLR-005.

LLR_016 *Maintain Track File*: The TrafficTracking shall maintain the track of a detected traffic element until it is no longer detected or poses no threat.
Traceability: Developed from HLR-005.

LLR_017 *Track File Accuracy*: The TrafficTracking shall ensure that the Position and VelocityVector in the track file have an accuracy within Y meters and Z meters/second respectively.
Traceability: Developed from HLR-005.

LLR_018 *Collision Potential Calculation*: The CollisionEvaluation shall calculate the collision potential for each tracked traffic element using Position, VelocityVector, and TrajectoryData.
Traceability: Developed from HLR-006.

LLR_019 *Threat Level Assessment*: The CollisionEvaluation shall assess the threat level of each traffic element based on the calculated collision potential.
Traceability: Developed from HLR-006.

LLR_020 *Time to Collision Calculation*: The ThreatPrioritization shall calculate the TimeToCollision (TTC) for each tracked traffic element.
Traceability: Developed from HLR-007.

LLR_021 *Threat Ranking*: The ThreatPrioritization shall rank traffic elements based on their TimeToCollision, prioritizing the ones with the shortest TTC.
Traceability: Developed from HLR-007.

LLR_022 *Avoidance Maneuver Calculation*: The ManeuverDetermination shall calculate an avoidance maneuver for each identified collision threat.
Traceability: Developed from HLR-008.

LLR_023 *Maneuver Selection*: The ManeuverDetermination shall select the most appropriate avoidance maneuver from the calculated options.
Traceability: Developed from HLR-008.

LLR_08a-01 *Maneuver Assignment for Identified Threats*: The ManeuverDetermination function shall assign a specific avoidance maneuver to each identified collision threat unless a higher-priority threat takes precedence.
Traceability: Developed from DHLR-008a.

LLR_08a-02 *Override Handling for Higher-Priority Threats*: If a higher-priority threat is detected after an initial maneuver is assigned, the CAS shall override the initial maneuver and activate the higher-priority maneuver.
Traceability: Developed from DHLR-008a.

LLR_024 *Command Maneuver*: The ManeuverCommand function shall issue a command to initiate the selected avoidance maneuver.
Traceability: Developed from HLR-009.

LLR_025 *Continue Maneuver*: The ManeuverCommand function shall issue a command to continue an ongoing avoidance maneuver if the threat persists.
Traceability: Developed from HLR-009.

LLR_026 *Terminate Maneuver*: The ManeuverCommand function shall issue a command to terminate an ongoing avoidance maneuver if the threat no longer exists.
Traceability: Developed from HLR-009.

LLR_008b-01 *Real-Time Monitoring of Active Maneuvers*: The CAS shall monitor each active maneuver in real-time, evaluating whether the associated threat still exists or if new threats emerge.
Traceability: Developed from DHLR-008b.

LLR_008b-02 *Maneuver Update Based on Threat Reassessment*: If the threat assessment changes, the CAS shall update the maneuver accordingly. This update can involve adjusting the existing maneuver or initiating a new one.
Traceability: Developed from DHLR-008b.

LLR_008b-03 *Termination of Maneuvers*: The CAS shall terminate an active maneuver when the associated threat is no longer detected or when the maneuver is superseded by a higher-priority action.
Traceability: Developed from DHLR-008b.

LLR_027 *Sensor Data Integrity Check*: The CAS shall perform integrity checks on sensor data before using it for detection and tracking.
Traceability: Developed from HLR-010.

LLR_028 *Startup Self-Test*: The CAS shall perform a self-test on startup to ensure all components are functioning correctly.
Traceability: Developed from HLR-011.

LLR_029 *Periodic Self-Test*: The CAS shall perform self-tests periodically during operation to ensure ongoing functionality of all components.
Traceability: Developed from HLR-011.

LLR_030 *Redundancy Strategy Implementation*: The CAS shall implement redundancy strategies to handle failures in primary detection and tracking components.
Traceability: Developed from HLR-012.

LLR_031 *Real-Time Communication*: The CAS shall transmit status and alerts to the ground control station in real-time.
Traceability: Developed from HLR-013.

LLR_032 *False Alarm Detection and Mitigation*: The CAS shall implement false alarm detection and mitigation strategies to minimize unnecessary maneuvers.
Traceability: Developed from HLR-014.

Software Architecture:

- Software subsystem: Traffic Detection
- Software Allocation: DETECT_TRAFFIC

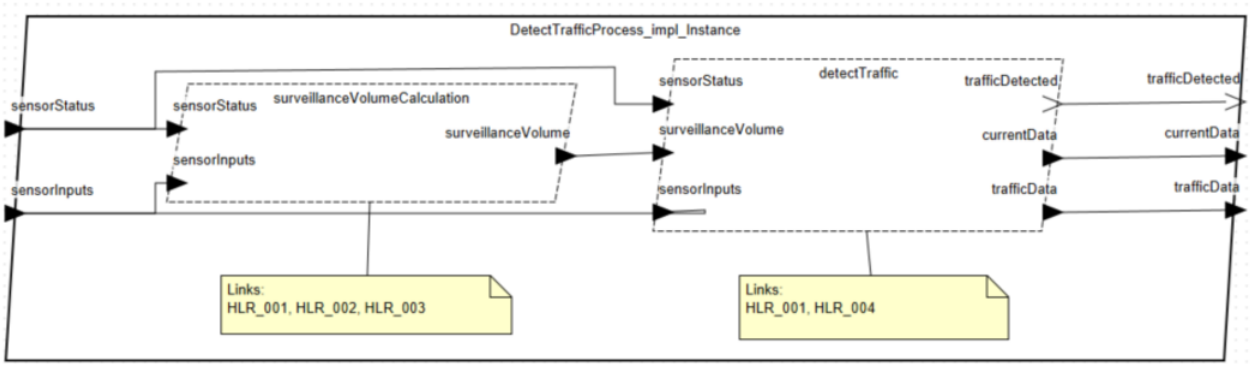

Figure A.1 AADL model of the traffic detection subsystem

- Software subsystem: Traffic Tracking
- Software Allocation: TRAFFIC_TRACKING

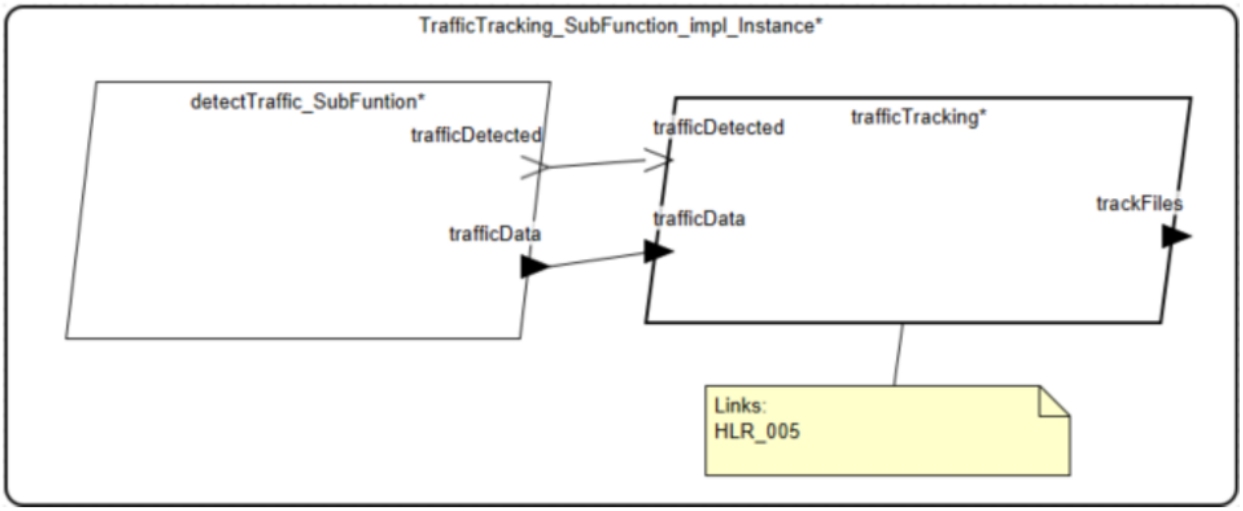

Figure A.2 AADL model of the traffic tracking subsystem

- Software subsystem: Collision Evaluation
- Software Allocation: COLLISION_EVALUATION

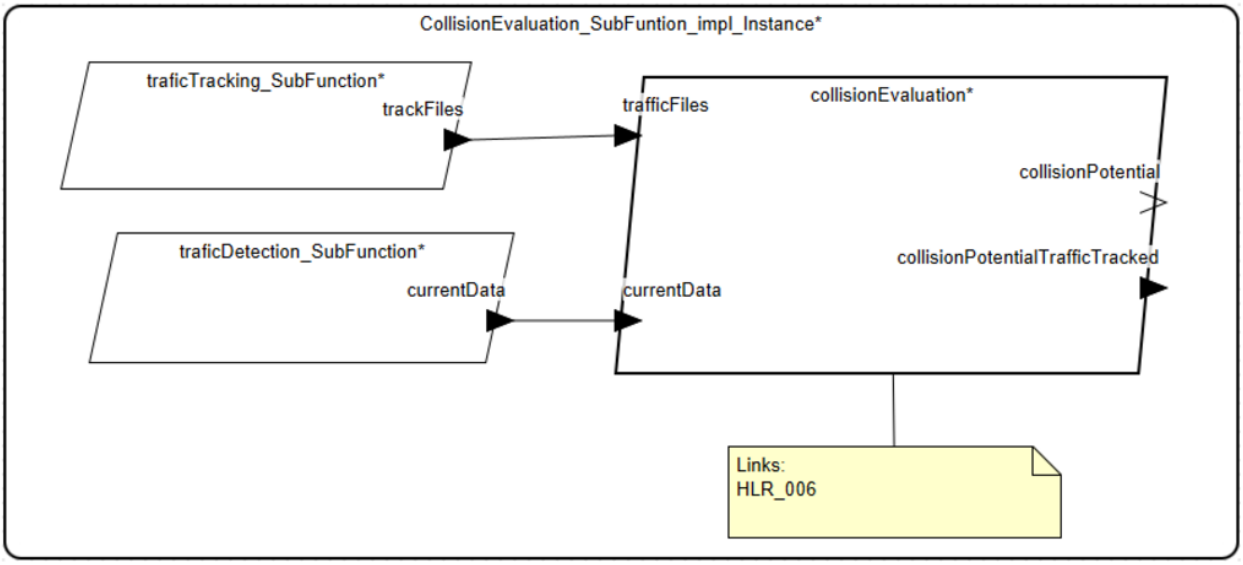

Figure A.3 AADL model of the collision evaluation subsystem

- Software subsystem: Threat Prioritization
- Software Allocation: THREAT_PRIORIZATION

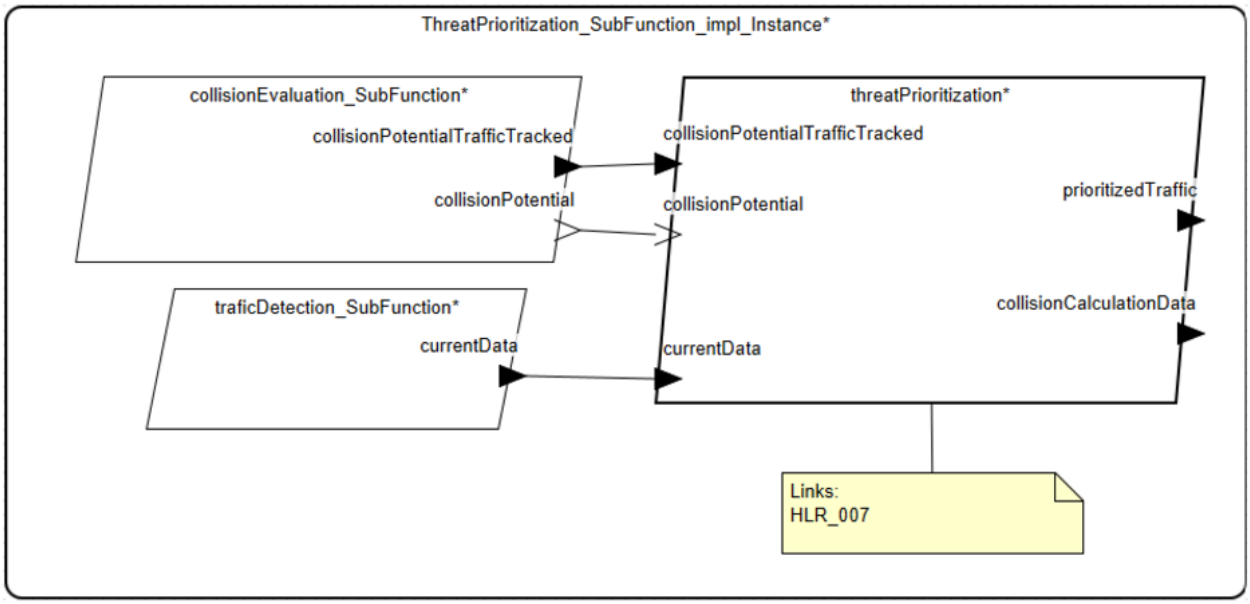

Figure A.4 AADL model of the threat prioritization subsystem

- Software subsystem: Maneuver Determination

- Software Allocation: MANEUVER_DETERMINATION

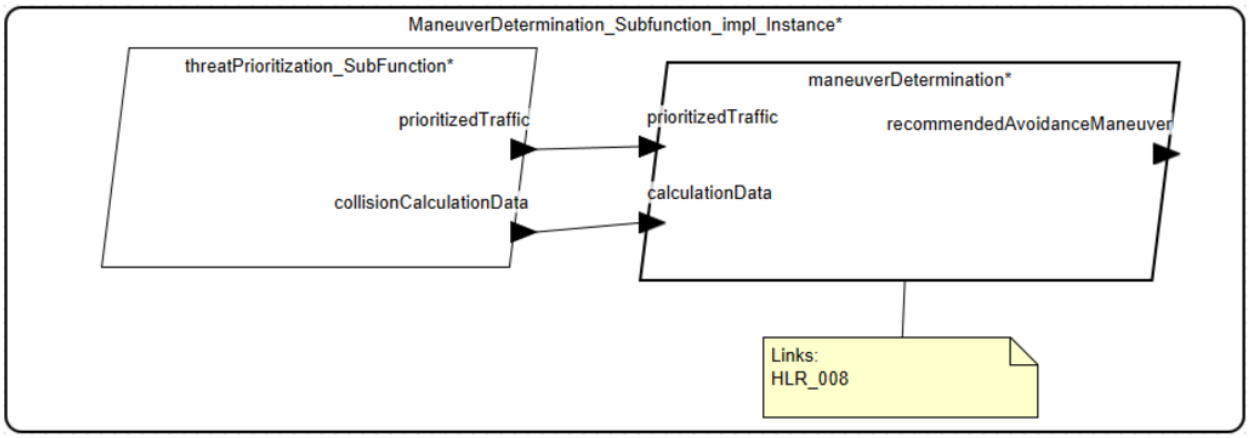

Figure A.5 AADL model of the maneuver determination subsystem

- Software subsystem: Maneuver Command
- Software Allocation: MANEUVER_COMMAND

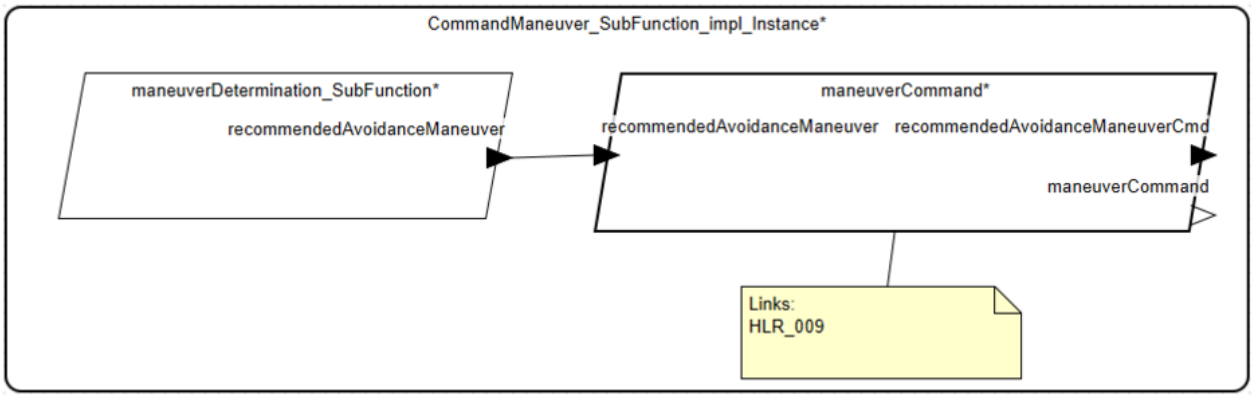

Figure A.6 AADL model of the maneuver command subsystem

The Failure Mode and Effects Analysis (FMEA) helped identify potential failures in each of the software's subsystems.

FMEA report:

Software Element	Failure Mode	Local Effect	Potentia Severity	Recommendation
Traffic Detection	Sensor Failure	No traffic detected	Critical (10)	Implement dual-sensing technology with automatic failover. Ensure periodic testing of failover capability.
Traffic Detection	Software Error	Incorrect traffic data processing	High (9)	Conduct extensive integration testing and add error-checking algorithms to validate data before processing.
Traffic Detection	Communication Failure	Delayed or no data transmission	High (8)	Implement redundant communication channels and improve error-handling procedures to detect and reroute data transmission automatically.
Traffic Detection	Configuration Error	Incorrect detection parameters set	Medium (7)	Standardize configuration procedures and introduce periodic configuration audits and validation tests.
Traffic Detection	Physical Obstruction	Sensor input blocked or degraded	Medium (6)	Schedule regular maintenance checks and install environmental monitoring systems to alert staff to obstructions.
Traffic Tracking	Loss of tracking data	Inability to continue monitoring detected traffic	High (8)	Implement data recovery protocols and redundant tracking mechanisms with automatic switching to backup systems.
Traffic Tracking	Erroneous velocity vector computation	Incorrect assessment of traffic movement	Medium (7)	Enhance algorithm accuracy with machine learning techniques and introduce additional validation checks at each computation stage.

Traffic Tracking	Sensor misalignment	Degraded tracking accuracy	Medium (6)	Implement regular calibration protocols and real-time monitoring of sensor alignment with alerts for deviations.
Traffic Tracking	Communication delay	Late update of traffic data, potential for outdated tracking info	High (8)	Upgrade to faster data transfer technology and establish a secondary communication protocol for redundancy.
Traffic Tracking	Software crash	System stops tracking traffic altogether	High (9)	Develop a robust error handling framework and system recovery processes including auto-restart features.
Collision Evaluation	Incorrect threat assessment	Incorrect collision potential evaluation	High (8)	Integrate comprehensive machine learning algorithms to refine threat assessment based on real-world data continuously.
Collision Evaluation	Delayed data processing	Late collision threat response	High (8)	Optimize processing algorithms for efficiency and upgrade hardware for faster data handling capabilities.
Collision Evaluation	Sensor data corruption	Inaccurate threat data used for evaluations	High (8)	Implement stringent data integrity checks and use error-correcting codes for all incoming data streams.
Collision Evaluation	Software malfunction	System fails to evaluate collision threats	Critical (10)	Establish a routine for periodic software audits and develop robust error handling and recovery procedures.

Collision Evaluation	Communication failure with tracking component	No updated data received for evaluation	High (8)	Ensure reliable communication protocols are in place and implement fallback mechanisms for immediate switchover when failures are detected.
Threat Prioritization	Incorrect ranking logic	Misordered threat prioritization	High (8)	Audit and optimize ranking algorithms annually and incorporate adaptive machine learning to enhance decision-making.
Threat Prioritization	Delay in processing updates	Outdated threat prioritization	High (8)	Invest in faster processing hardware and optimize real-time data handling capabilities to process updates instantaneously.
Threat Prioritization	Data synchronization errors	Inconsistent threat data used	High (8)	Implement a robust protocol for data verification and synchronization across all system components.
Threat Prioritization	Software glitches	Interruption in threat prioritization process	Critical (10)	Conduct frequent software testing and ensure updates are deployed; establish fail-safe modes for glitch detection.
Threat Prioritization	Communication breakdown with collision evaluation system	Lack of input for prioritization decisions	High (8)	Design a multi-channel communication strategy with automatic failover to backup channels in case of failure.
Maneuver Determination	Inaccurate threat assessment	Incorrect maneuver determination	High (9)	Continuously update and refine threat assessment algorithms based on latest data and simulation outcomes.

Maneuver Determination	Delay in maneuver computation	Delayed response to collision threats	High (9)	Streamline computation algorithms and invest in advanced computing hardware to minimize response times.
Maneuver Determination	Software bugs or glitches	Incorrect or no maneuver generated	Critical (10)	Implement comprehensive testing protocols, regular software audits, and updates to detect and rectify bugs.
Maneuver Determination	Communication latency with sensors	Outdated data used for maneuver planning	High (8)	Enhance the real-time data transmission system and implement a checking mechanism to confirm data freshness.
Maneuver Determination	System overload	System unresponsive or slow to determine maneuvers	Critical (10)	Design the system for scalability, implement load balancing, and prioritize critical data processing.
Maneuver Command	Command signal failure	No maneuver initiated or incorrect maneuver initiated	Critical (10)	Implement comprehensive signal integrity checks and establish multiple fallback communication protocols.
Maneuver Command	Delay in command execution	Delayed response in executing maneuvers	High (9)	Optimize system response time with real-time processing enhancements and streamline command execution paths.
Maneuver Command	Incorrect command data	Inappropriate maneuver based on incorrect or corrupted data	Critical (10)	Strengthen data validation processes at each stage and employ redundant data verification systems.
Maneuver Command	Software malfunction	Incorrect maneuver execution, potential system crash	Critical (10)	Establish a protocol for regular software testing, updates, and robust error handling systems.

Maneuver Command	Hardware failure	Inability to execute commands due to hardware malfunction	High (9)	Utilize high-reliability hardware components, conduct regular maintenance, and implement hardware health monitoring.

APPENDIX B REDAML: DEVELOPMENT AND CASE STUDY

This appendix shows the development of the DO-178C-compliant high-level requirements specification modelling language ReDaML. It will present aspects of the development of the Eclipse Obeo platform, as well as the data used and models created during the case study.

The abstract syntax represents the core of the modelling language. In addition to the enumeration definitions (Figure B.4), the meta-model has been divided into three parts to visualize it better. Figure B.1 describes the relationship between a requirement's specifications and the objectives and information of the DO-178C, such as the possibility of traceability with SRATS, justifications for including a derived requirement, or even justifications for deviation from the initial planning.

Figure B.2 was specified to allow the definition of interface and communication requirements between components and their characteristics. Figure B.3 shows the aspects related to safety. This allows the user to model safety features such as health monitoring.

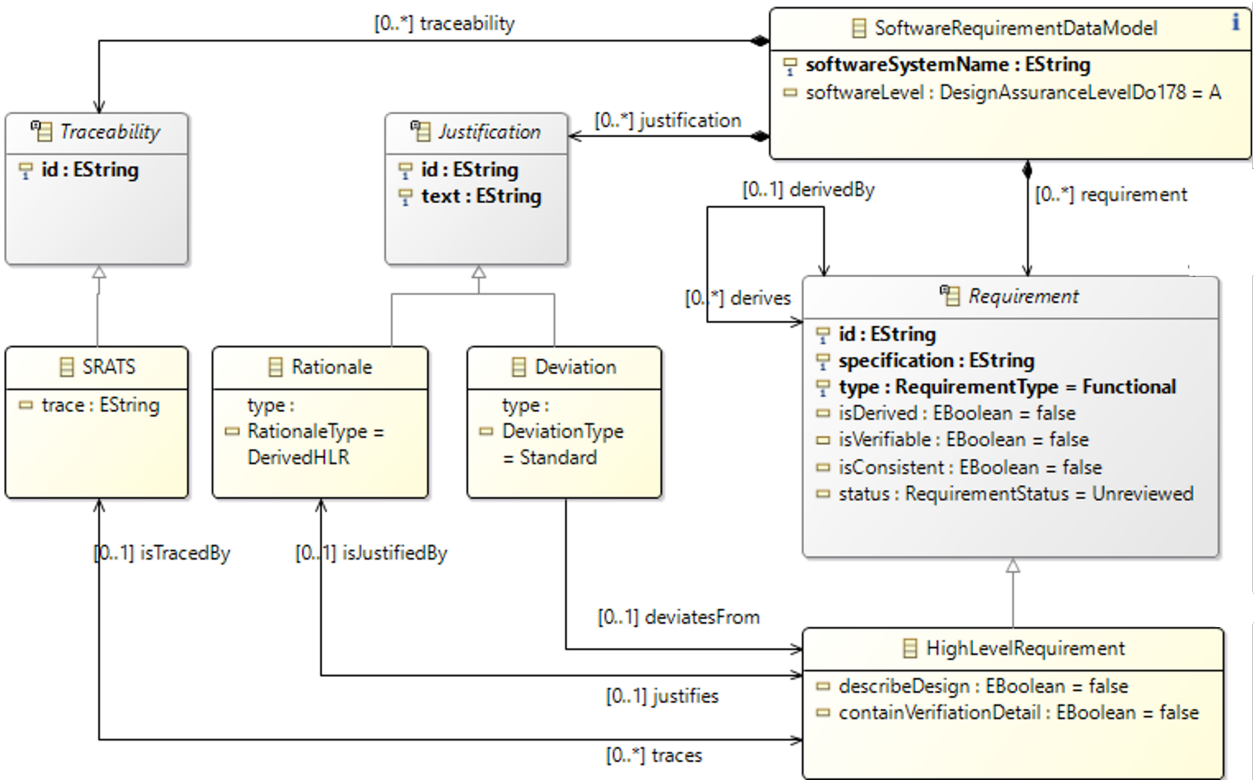

Figure B.1 The complete ReDaML meta-model

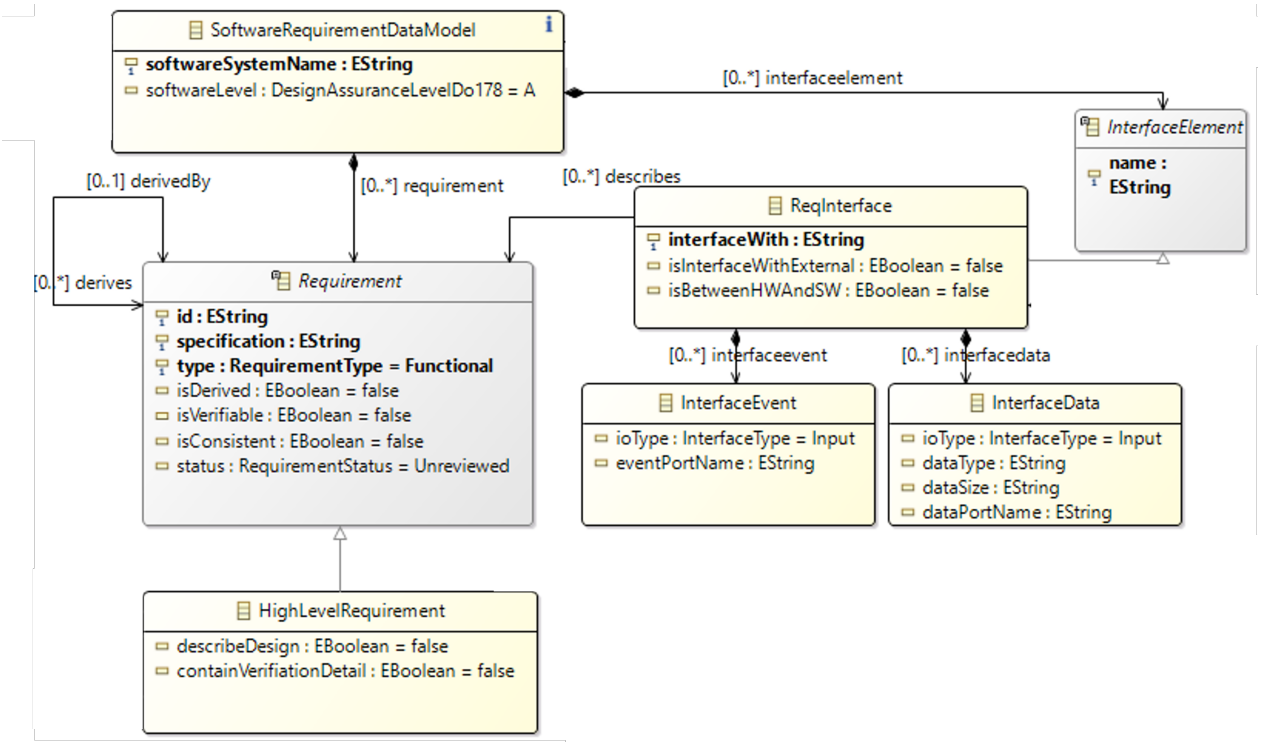

Figure B.2 The complete ReDaML meta-model

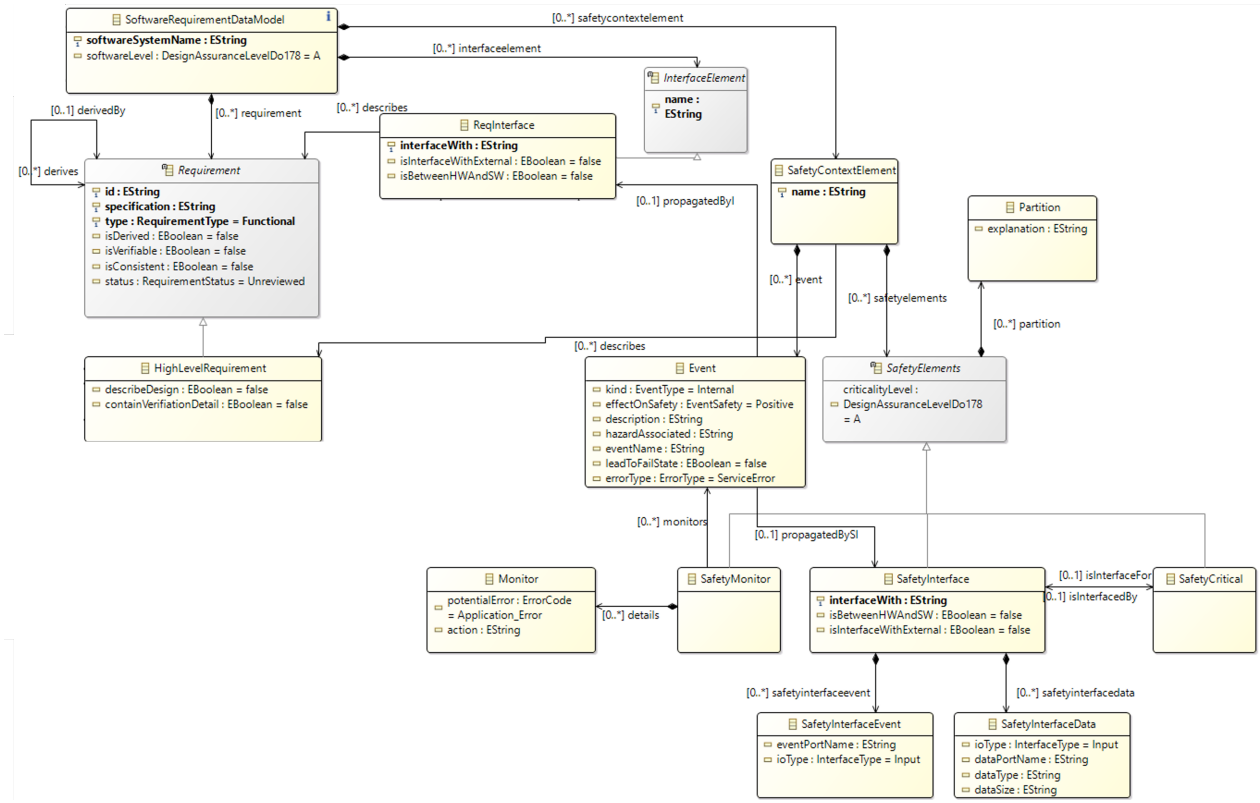

Figure B.3 The complete ReDaML meta-model

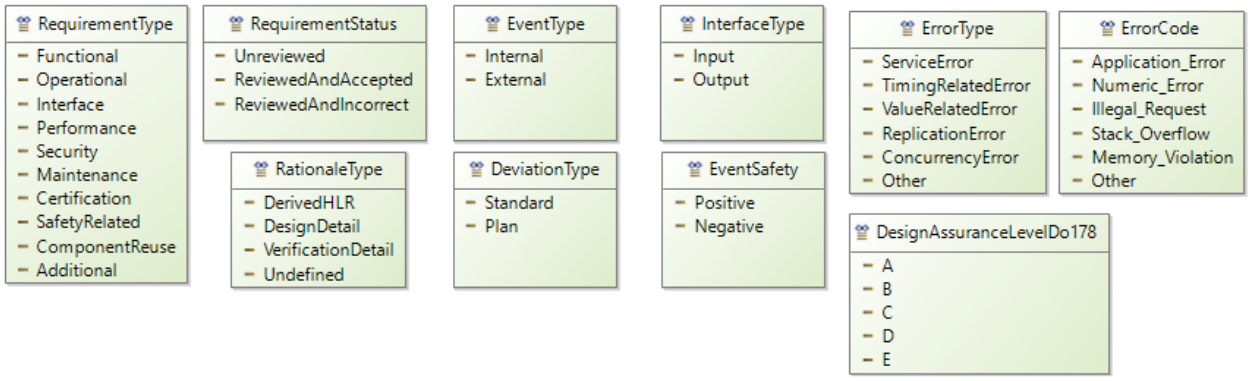

Figure B.4 The complete ReDaML meta-model

Concrete Syntax:

The concrete syntax defines how users visually and textually interact with the modelling environment. To facilitate and enhance intuitive modelling, ReDaML provides multiple ways to represent and manipulate the model's elements: diagrams, tables, and a tree. The diagrammatic notation allows users to create and visualize relationships between requirements, their

associated aspects, and traceability links in a structured graphical format. This approach has its elements defined through the project depicted in Figure B.5.

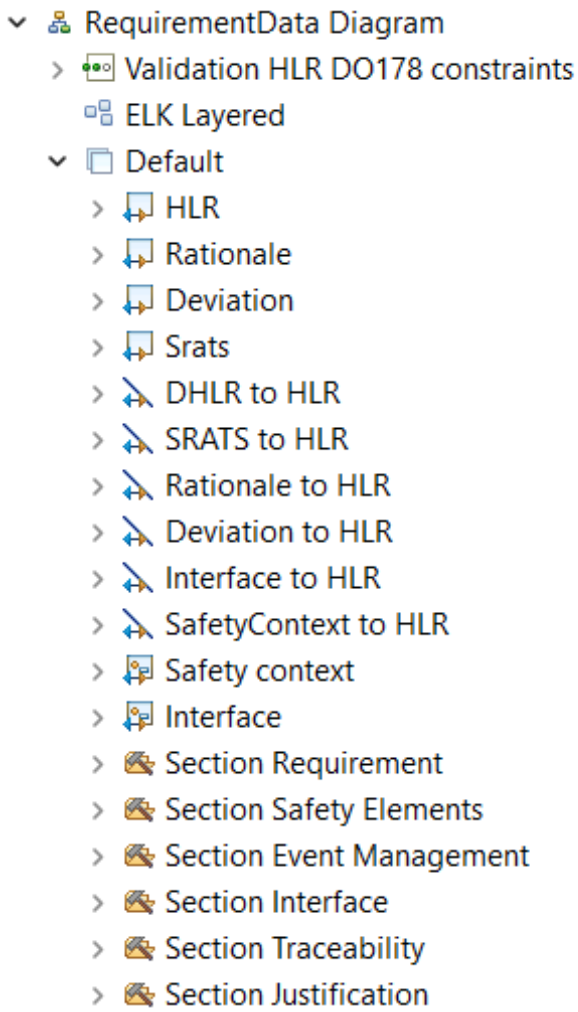

Figure B.5 Concrete syntax specification model - diagram view

In addition to the graphical representation, a table-based syntax was implemented to enable structured data entry and validation. This allows users to systematically define requirement attributes, traceability links, and verification information in a format that supports quick editing and structured validation. Figure B.6 illustrates the specification view in the project.

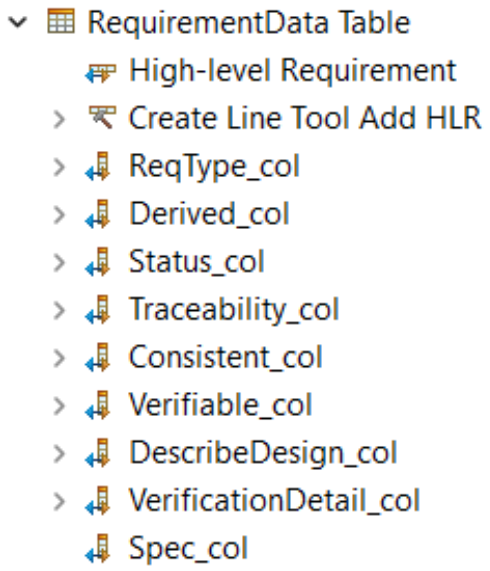

Figure B.6 Concrete syntax specification model - table view

The tree-based representation specification (Figure B.7) supports hierarchical modelling created explicitly for the safety aspects of the model, where users can navigate through the model's structure, organizing elements in a way that reflects their logical dependencies.

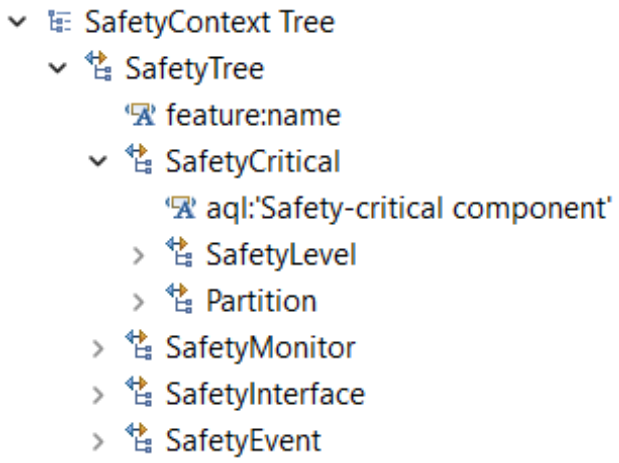

Figure B.7 Concrete syntax specification model - tree view

ReDaML Semantics:

Object Constraint Language (OCL) constraints are used to formally define the semantics of ReDaML, guaranteeing that the models created by the user follow the intended structure and validation rules. Constraints are expressed precisely and declaratively with OCL, enabling

model consistency and domain-specific rule compliance checks. By preventing inconsistencies and guaranteeing adherence to DO-178C objectives and information, these constraints ensure that each modelled element conforms to the specified structure and logic. Figure B.8 illustrates the *odesign* project setup in Obeo Designer, where OCL constraints were defined to ensure the correctness of the DSML.

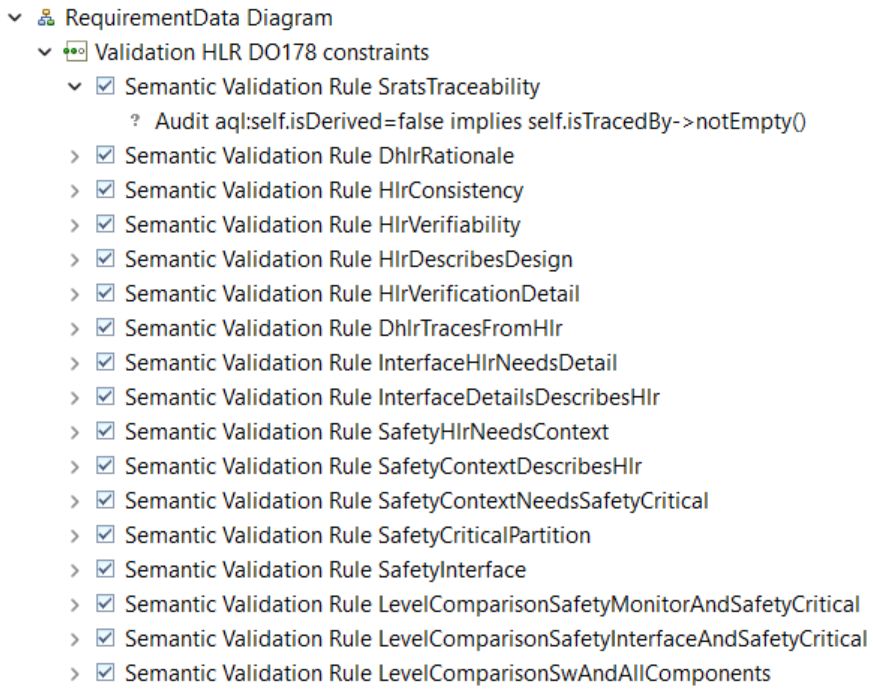

Figure B.8 ReDaML's semantics view in Obeo

The ReDaML rules defining its semantics are explained below, and the expression is shown in Table B.1. The *Problem Level* column represents the level of message raised when the rule is violated.

- **SratsTraceability:** Ensures traceability of that non-derived requirements are linked to at least one traceable element.
- **DhlrRationale:** Verifies that derived requirements are justified by at least one element.
- **HlrConsistency:** Checks if the high-level requirements (HLR) are internally consistent.
- **HlrVerifiability:** Ensures that each HLR can be verified through defined criteria.

- **HlrDescribesDesign:** Confirms that HLRs meant to describe design are properly justified by design elements.
- **HlrVerificationDetail:** Ensures that HLRs containing verification details are justified by relevant elements.
- **DhlrTracesFromHlr:** Validates that derived HLRs trace back to at least one source HLR.
- **InterfaceHlrNeedsDetail:** Verifies that interface-related requirements are linked to the appropriate interface details in the design.
- **InterfaceDetailsDescribesHlr:** Confirms that interface details describe at least one requirement.
- **SafetyHlrNeedsContext:** Ensures that safety-related requirements are associated with safety context elements in the design.
- **SafetyContextDescribesHlr:** Validates that safety context elements describe at least one safety requirement.
- **SafetyContextNeedsSafetyCritical:** Ensures that each safety context contains at least one safety-critical element.
- **SafetyCriticalPartition:** Confirms that safety-critical elements are assigned to appropriate partitions.
- **SafetyInterface:** Validates that each safety-critical context includes at least one safety interface element.
- **LevelComparisonSafetyMonitorAndSafetyCritical:** Checks that the safety monitor's criticality level is equal to or lower than the safety-critical element's level.
- **LevelComparisonSafetyInterfaceAndSafetyCritical:** Ensures that the criticality level of the safety interface is not higher than that of the linked safety-critical element.
- **LevelComparisonSwAndAllComponents:** Verifies that the software level does not exceed the criticality level of associated safety-critical elements.

Validation Name	Expression	Problem Level
SratsTraceability	`aql:self.isDerived=false implies self.isTracedBy->notEmpty()`	WARNING
DhlrRationale	`aql:self.isDerived = true implies self.isJustifiedBy->notEmpty()`	ERROR
HlrConsistency	`feature:isConsistent`	WARNING
HlrVerifiability	`feature:isVerifiable`	WARNING
HlrDescribesDesign	`aql:self.describeDesign = true implies self.isJustifiedBy->notEmpty()`	WARNING
HlrVerificationDetail	`aql:self.containVerificationDetail = true implies self.isJustifiedBy->notEmpty()`	WARNING
DhlrTracesFromHlr	`aql:self.isDerived = true implies self.derivedBy->notEmpty() or self.derives->notEmpty()`	WARNING
InterfaceHlrNeedsDetail	`aql:self.type = redaml::RequirementType::Interface implies self.eContainer().eAllContents( redaml::ReqInterface).describes->includes(self)`	ERROR
InterfaceDetailsDescribesHlr	`aql:self.describes->notEmpty()`	ERROR
SafetyHlrNeedsContext	`aql:self.type = redaml::RequirementType::SafetyRelated implies self.eContainer().eAllContents( redaml::SafetyContextElement).describes->includes(self)`	ERROR
SafetyContextDescribesHlr	`aql:self.describes->notEmpty()`	ERROR
SafetyContextNeeds-SafetyCritical	`aql:self.name+' must contains at least one safety-critical element'`	ERROR
SafetyCriticalPartition	`aql:self.partition->notEmpty()`	ERROR
SafetyInterface	`aql:self.eContents()->filter(redaml::SafetyInterface)-->notEmpty()`	ERROR
LevelComparisonSafety-MonitorAndSafetyCritical	`aql:self.eContainer().safetyelements->forAll( sc: redaml::SafetyCritical | self.criticalityLevel <= sc.criticalityLevel)`	ERROR
LevelComparisonSafety-InterfaceAndSafetyCritical	`aql:self.criticalityLevel<= self.isInterfaceFor.criticalityLevel`	ERROR
LevelComparisonSwAnd-AllComponents	`aql:self.safetycontextelement.safetyelements->forAll( sc: redaml::SafetyCritical | self.softwareLevel <= sc.criticalityLevel)`	ERROR

Table B.1 Validation rules with corresponding expressions and problem levels

The UAV Collision Avoidance Case Study This case study demonstrates how the HLRs can be represented and validated using the ReDaML to adhere to DO-178C objectives. The UAV Collision Avoidance system was chosen, and it was inspired by [94]. The entire set of HLRs utilized by the system is presented in Table B.2.

ID	**Description**	**Type**	**Trace from**
HLR-1	The CA subsystem software shall process sensor data and detect traffic within its surveillance volume.	Functional	SRATS-1: The Collision Avoidance System shall detect and avoid conflicting traffic.
DHLR-1.1	The CAS shall detect traffic with sufficient time remaining for the successful performance of all required collision avoidance functions.	Safety	SRATS-1
DHLR-1.2	The CA subsystem software shall detect traffic at a range of at least 20 nautical miles	Functional	SRATS-1
DHLR-1.3	The CA subsystem software shall detect traffic within an azimuth FOR of at least +/-110° referenced from the flight path of the UAV.	Functional	SRATS-1
DHLR-1.4	The CA subsystem software shall detect traffic within an elevation FOR of at least +/-15° referenced from the flight path of the UAV.	Functional	SRATS-1
DHLR-1.5	The CA subsystem software shall detect traffic with accuracy in range, altitude, and azimuth determinations that are at least equivalent to the TCAS-II system.	Safety	SRATS-1
DHLR-1.6	False detections shall occur at a rate that supports the false track guideline.	Performance	SRATS-1
HLR-2	The CA subsystem software shall continuously track the detected traffic objects.	Functional	SRATS-2: The Collision Avoidance System shall track the detected traffic.

Continued on next page...

ID	Description	Type	Trace from
DHLR-2.1	The CA subsystem software shall track traffic within an azimuth FOR of at least +/-110° and elevation FOR of at least +/-15° referenced from the flight path of the UAV.	Functional	SRATS-2
DHLR-2.2	The CA subsystem software shall be capable of simultaneously maintaining tracks on at least 35 cooperative aircraft in the UA platform's surveillance volume.	Performance	SRATS-2
DHLR-2.3	The CA subsystem software shall establish a unique track on at least 95% of the cooperative traffic within the UA platform's surveillance volume.	Performance	SRATS-2
DHLR-2.4	The CA subsystem software shall establish and maintain a track for traffic deemed eligible for collision avoidance with a confidence of 95%.	Performance	SRATS-2
DHLR-2.5	The CA subsystem software shall maintain track accuracy that is sufficient to support the requirement to detect collision threats, and to not generate nuisance alerts on non-threats.	Safety	SRATS-2
DHLR-2.6	False tracks shall account for no more than 1.2% of all tracks established by the CA subsystem software.	Safety	SRATS-2
DHLR-2.7	The CA subsystem software shall establish a track on detected traffic in the surveillance volume within 12.5 seconds of initial detection.	Safety	SRATS-2

Continued on next page...

ID	Description	Type	Trace from
HLR-3	The CA subsystem software shall evaluate the relative positions, velocities, and trajectories of each traffic element being tracked to determine the potential for collision.	Functional	SRATS-3: The Collision Avoidance System shall evaluate the potential for collision with each traffic element being tracked, including the assessment of existing collision threats.
DHLR-3.1	The CA subsystem software shall continuously determine if any detected traffic elements pose a collision threat to the UA.	Safety	SRATS-3
DHLR-3.2	The CA subsystem software shall determine the potential for collision by each tracked traffic element within 5 seconds of track establishment.	Safety	SRATS-3
DHLR-3.3	The CA subsystem software shall provide real-time alerts, warnings, or recommendations to the UAV operator or autonomous flight control system regarding the presence of collision threats.	Safety	SRATS-3
HLR-4	The CA subsystem software shall assign a priority level to each tracked traffic element based on the assessed collision threat.	Functional	SRATS-4: The Collision Avoidance System shall prioritize the traffic posing a collision threat.
DHLR-4.1	The CA subsystem software shall prioritize every traffic element that has been deemed a collision threat.	Safety	SRATS-4
DHLR-4.2	The CA subsystem software shall prioritize the collision threats within 4 seconds of collision potential classification.	Performance	SRATS-4

Continued on next page...

ID	Description	Type	Trace from
HLR-5	The CA subsystem software shall generate actionable avoidance commands that direct the UAV to perform the necessary maneuvers, ensuring effective collision prevention.	Functional	SRATS-5: The Collision Avoidance System shall determine an avoidance maneuver that prevents a collision.
DHLR-5.1	The CA subsystem software shall determine a feasible avoidance maneuver that prevents a collision with the most immediate threat from occurring.	Safety	SRATS-5
DHLR-5.2	The rate at which the CA subsystem provides maneuver guidance that creates a more hazardous situation shall be on the order of 1.0X10-6 per flight hour.	Safety	SRATS-5
DHLR-5.3	The probability of an unnecessary evasion maneuver advisory being generated without failure annunciation shall be on the order of 1X10-4 per flight hour in the terminal environment and 1.0X10-5 per flight hour in the en route environment.	Safety	SRATS-5
DHLR-5.4	The CA subsystem software shall provide real-time guidance and commands to the UAV autonomous flight control system, indicating the required actions to perform the avoidance maneuver and prevent a collision.	Safety	SRATS-5
DHLR-5.5	The CA subsystem software shall notify the ground station system of the recommended avoidance maneuver with a minimum of 1 second remaining to initiate the maneuver and avoid collision.	Safety	SRATS-5

Continued on next page...

ID	Description	Type	Trace from
HLR-6	The CA subsystem software shall command the appropriate avoidance maneuver to the UAV autonomous flight control system	Functional	SRATS-6: The Collision Avoidance System shall command an appropriate avoidance maneuver.
DHLR-6.1	The UAS shall command an avoidance maneuver within 4 seconds of a collision avoidance maneuver being advised by the CAS.	Safety	SRATS-6
HLR-7	The CA subsystem shall output maneuver commands to the flight management system.	Interface	System architecture, and Interface Control Document.
HLR-8	The CA subsystem shall acquire sensor data and information from the sensor fusion system.	Interface	System architecture, and Interface Control Document.
HLR-9	The CA subsystem shall interact with the guidance system to obtain information about the UAV's position, speed, altitude, and heading.	Interface	System architecture, and Interface Control Document.
HLR-10	The CA subsystem shall interact with the Ground Control Station, which serves as the command and control center for the UAV.	Interface	System architecture, and Interface Control Document.

Table B.2 High-Level Requirements (HLRs) for the Collision Avoidance System

The HLRs described above were modelled using ReDaML, and the three views provided by the language are depicted: diagram (Figures B.9, B.10, B.11, B.13 and B.14), table (Figure B.16) and tree (Figure B.17).

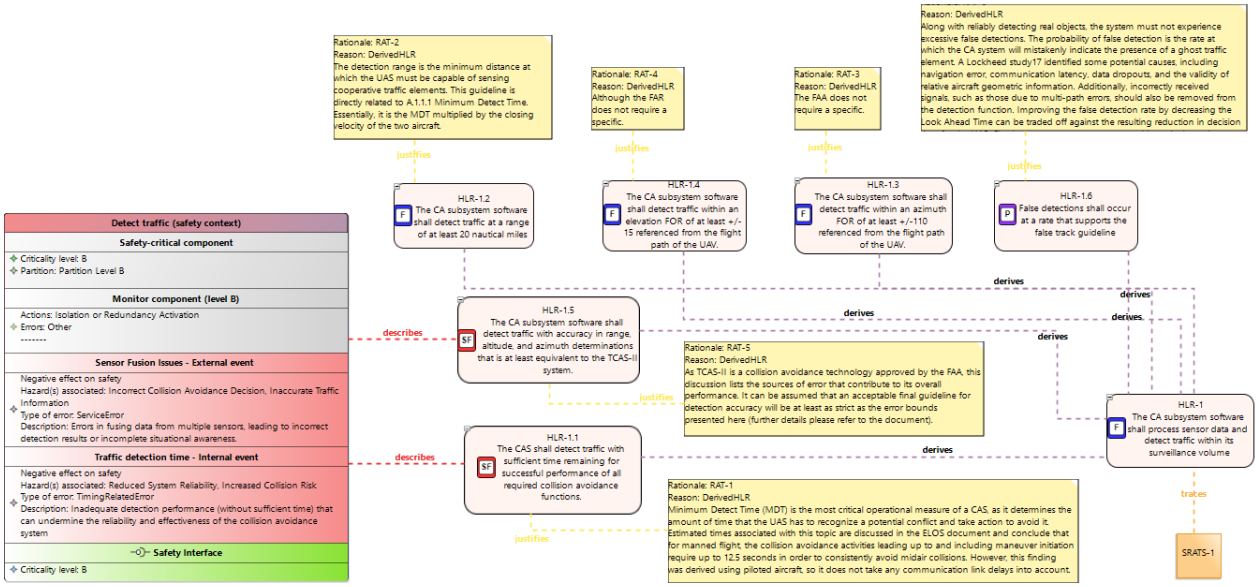

Figure B.9 HLR-1 and its DHLR modelled with ReDaML - diagram view

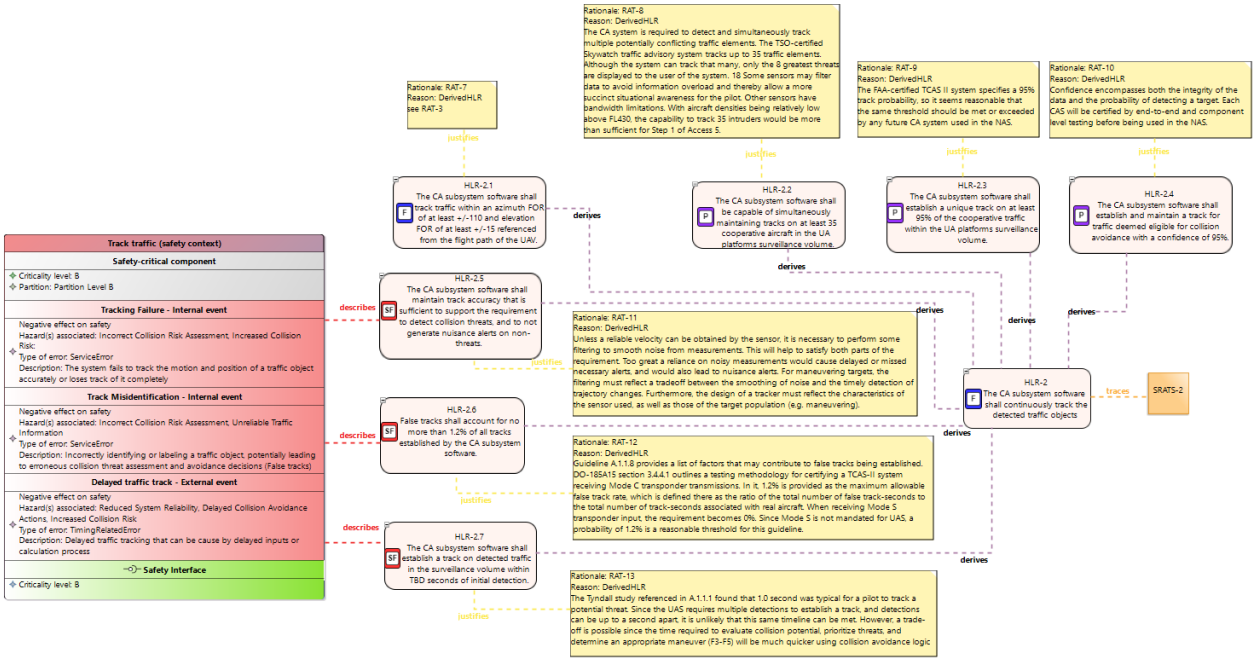

Figure B.10 HLR-2 and its DHLR modelled with ReDaML - diagram view

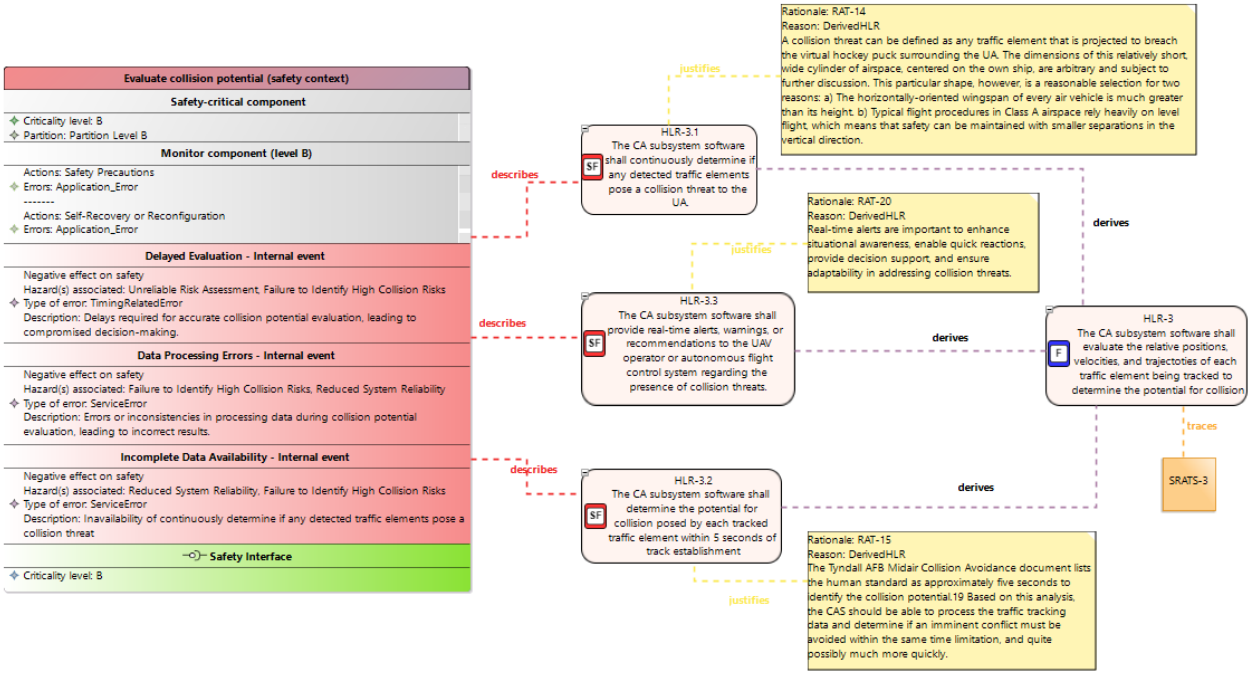

Figure B.11 HLR-3 and its DHLR modelled with ReDaML - diagram view

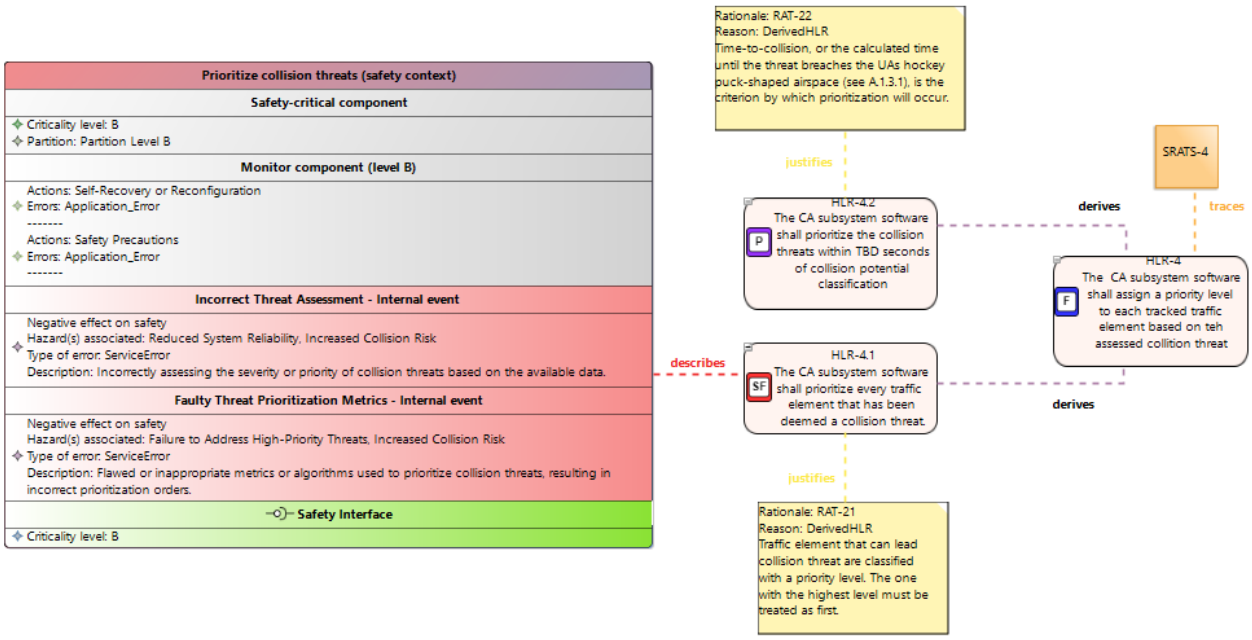

Figure B.12 HLR-4 and its DHLR modelled with ReDaML - diagram view

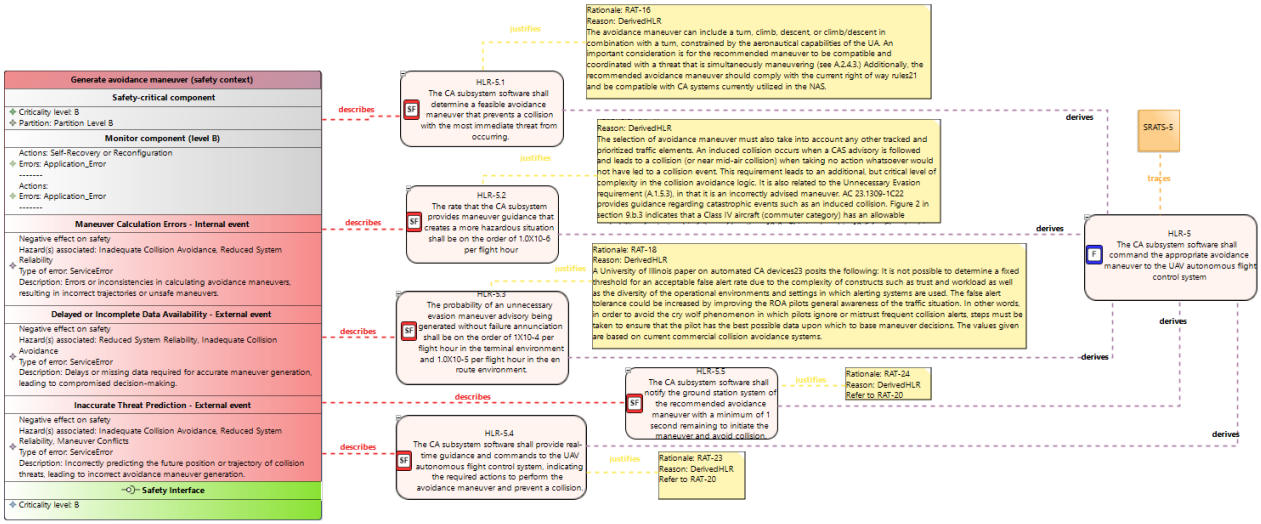

Figure B.13 HLR-5 and its DHLR modelled with ReDaML - diagram view

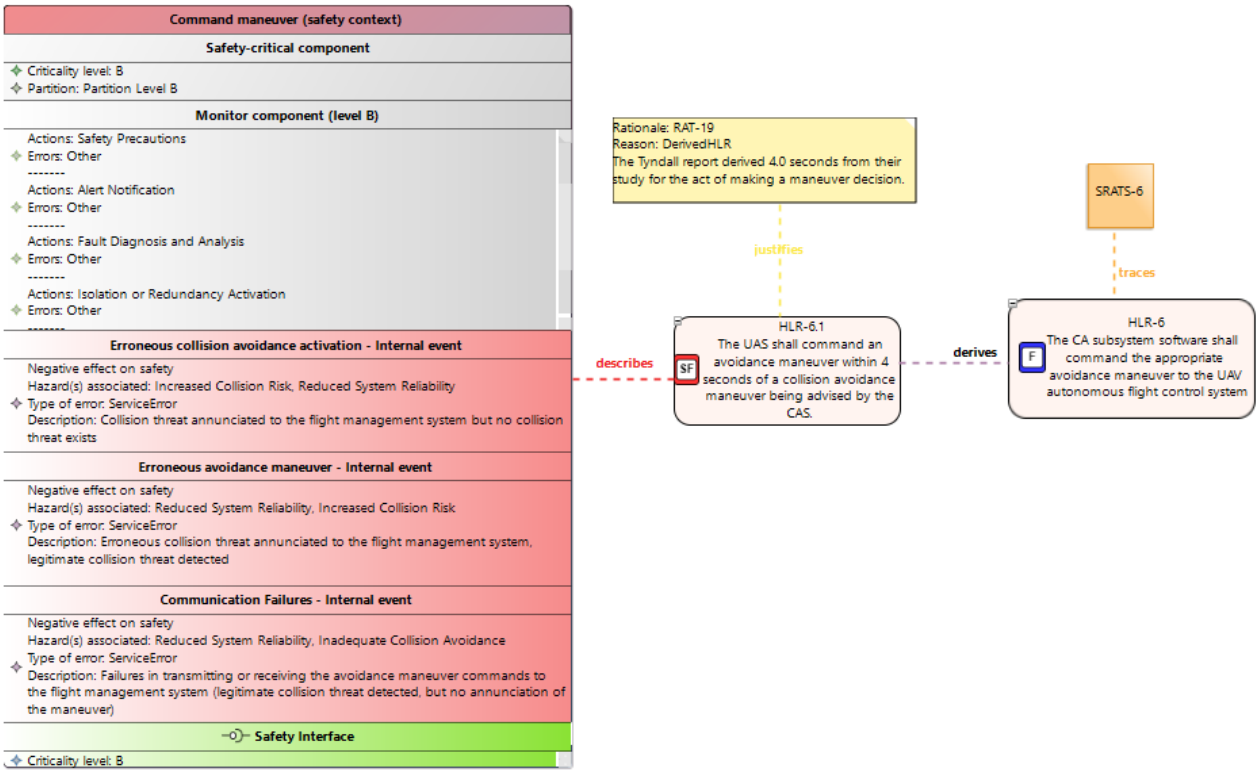

Figure B.14 HLR-6 and its DHLR modelled with ReDaML - diagram view

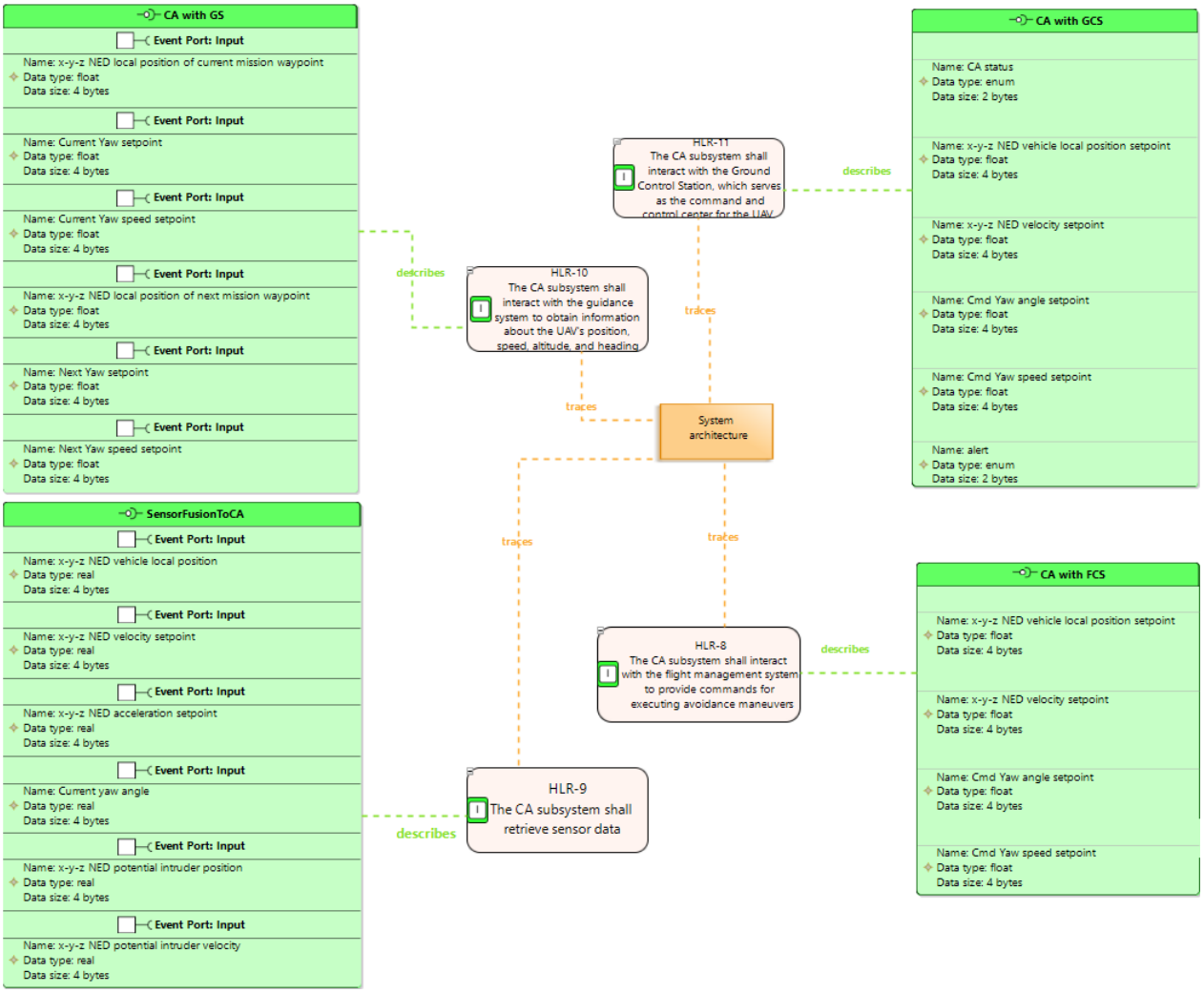

Figure B.15 HLR-8, HLR-9, HLR-10 and HLR-11 modelled with ReDaML - diagram view

	Type	Derived HLR	Review status	Traceability	Consistent	Verifiable	Describe Design	Contain Verification Details	Specification
HLR-1	Functional	☐	ReviewedAndAccepted	SRATS-1	☐ Yes	☐ Yes	☐ No	☐ No	The CA subsystem software shall process sensor data and detect traffic within its surveillance volume
HLR-2	Functional	☐	ReviewedAndAccepted	SRATS-2	☐ Yes	☐ Yes	☐ No	☐ No	The CA subsystem software shall continuously track the detected traffic objects
HLR-3	Functional	☐	ReviewedAndAccepted	SRATS-3	☐ Yes	☐ Yes	☐ No	☐ No	The CA subsystem software shall evaluate the relative positions, velocities, and trajectoties of each traffic element being tracked to det
HLR-4	Functional	☐	ReviewedAndAccepted	SRATS-4	☐ Yes	☐ Yes	☐ No	☐ No	The CA subsystem software shall assign a priority level to each tracked traffic element based on teh assessed collition threat
HLR-5	Functional	☐	ReviewedAndAccepted	SRATS-5	☐ Yes	☐ Yes	☐ No	☐ No	The CA subsystem software shall command the appropriate avoidance maneuver to the UAV autonomous flight control system
HLR-1.1	SafetyRelated	☐ Yes	ReviewedAndAccepted	SRATS-1	☐ Yes	☐ Yes	☐ No	☐ No	The CAS shall detect traffic with sufficient time remaining for successful performance of all required collision avoidance functions.
HLR-1.2	Functional	☐ Yes	ReviewedAndAccepted	SRATS-1	☐ Yes	☐ Yes	☐ No	☐ No	The CA subsystem software shall detect traffic at a range of at least 20 nautical miles
HLR-1.3	Functional	☐ Yes	ReviewedAndAccepted	SRATS-1	☐ Yes	☐ Yes	☐ No	☐ No	The CA subsystem software shall detect traffic within an azimuth FOR of at least +/-110 referenced from the flight path of the UAV.
HLR-1.4	Functional	☐ Yes	ReviewedAndAccepted	SRATS-1	☐ Yes	☐ Yes	☐ No	☐ No	The CA subsystem software shall detect traffic within an elevation FOR of at least +/-15 referenced from the flight path of the UAV.
HLR-1.5	SafetyRelated	☐ Yes	ReviewedAndAccepted	SRATS-1	☐ Yes	☐ Yes	☐ No	☐ No	The CA subsystem software shall detect traffic with accuracy in range, altitude, and azimuth determinations that is at least equivalent t
HLR-1.6	Performance	☐ Yes	ReviewedAndAccepted	SRATS-1	☐ Yes	☐ Yes	☐ No	☐ No	False detections shall occur at a rate that supports the false track guideline
HLR-2.1	Functional	☐ Yes	ReviewedAndAccepted	SRATS-2	☐ Yes	☐ Yes	☐ No	☐ No	The CA subsystem software shall track traffic within an azimuth FOR of at least +/-110 and elevation FOR of at least +/-15 referenced f
HLR-2.2	Performance	☐ Yes	ReviewedAndAccepted	SRATS-2	☐ Yes	☐ Yes	☐ No	☐ No	The CA subsystem software shall be capable of simultaneously maintaining tracks on at least 35 cooperative aircraft in the UA platform
HLR-2.3	Performance	☐ Yes	ReviewedAndAccepted	SRATS-2	☐ Yes	☐ Yes	☐ No	☐ No	The CA subsystem software shall establish a unique track on at least 95% of the cooperative traffic within the UA platforms surveillance
HLR-2.4	Performance	☐ Yes	ReviewedAndAccepted	SRATS-2	☐ Yes	☐ Yes	☐ No	☐ No	The CA subsystem software shall establish and maintain a track for traffic deemed eligible for collision avoidance with a confidence of 9
HLR-2.5	SafetyRelated	☐ Yes	ReviewedAndAccepted	SRATS-2	☐ Yes	☐ Yes	☐ No	☐ No	The CA subsystem software shall maintain track accuracy that is sufficient to support the requirement to detect collision threats, and t
HLR-2.6	SafetyRelated	☐ Yes	ReviewedAndAccepted	SRATS-2	☐ Yes	☐ Yes	☐ No	☐ No	False tracks shall account for no more than 1.2% of all tracks established by the CA subsystem software.
HLR-2.7	SafetyRelated	☐ Yes	ReviewedAndAccepted	SRATS-2	☐ Yes	☐ Yes	☐ No	☐ No	The CA subsystem software shall establish a track on detected traffic in the surveillance volume within TBD seconds of initial detection
HLR-3.1	SafetyRelated	☐ Yes	ReviewedAndAccepted	SRATS-3	☐ Yes	☐ Yes	☐ No	☐ No	The CA subsystem software shall continuously determine if any detected traffic elements pose a collision threat to the UA.
HLR-3.2	SafetyRelated	☐ Yes	ReviewedAndAccepted	SRATS-3	☐ Yes	☐ Yes	☐ No	☐ No	The CA subsystem software shall determine the potential for collision posed by each tracked traffic element within 5 seconds of track e
HLR-3.3	SafetyRelated	☐ Yes	ReviewedAndAccepted	SRATS-3	☐ Yes	☐ Yes	☐ No	☐ No	The CA subsystem software shall provide real-time alerts, warnings, or recommendations to the UAV operator or autonomous flight con
HLR-4.1	SafetyRelated	☐ Yes	ReviewedAndAccepted	SRATS-4	☐ Yes	☐ Yes	☐ No	☐ No	The CA subsystem software shall prioritize every traffic element that has been deemed a collision threat.
HLR-4.2	Performance	☐ Yes	ReviewedAndAccepted	SRATS-4	☐ Yes	☐ Yes	☐ No	☐ No	The CA subsystem software shall prioritize the collision threats within TBD seconds of collision potential classification
HLR-5.1	SafetyRelated	☐ Yes	ReviewedAndAccepted	SRATS-5	☐ Yes	☐ Yes	☐ No	☐ No	The CA subsystem software shall determine a feasible avoidance maneuver that prevents a collision with the most immediate threat fro
HLR-5.2	SafetyRelated	☐ Yes	ReviewedAndAccepted	SRATS-5	☐ Yes	☐ Yes	☐ No	☐ No	The rate that the CA subsystem provides maneuver guidance that creates a more hazardous situation shall be on the order of 1.0X10-6
HLR-5.3	SafetyRelated	☐ Yes	ReviewedAndAccepted	SRATS-5	☐ Yes	☐ Yes	☐ No	☐ No	The probability of an unnecessary evasion maneuver advisory being generated without failure annunciation shall be on the order of 1X10
HLR-5.4	SafetyRelated	☐ Yes	ReviewedAndAccepted	SRATS-5	☐ Yes	☐ Yes	☐ No	☐ No	The CA subsystem software shall provide real-time guidance and commands to the UAV autonomous flight control system, indicating t
HLR-5.5	SafetyRelated	☐ Yes	ReviewedAndAccepted	SRATS-5	☐ Yes	☐ Yes	☐ No	☐ No	The CA subsystem software shall notify the ground station system of the recommended avoidance maneuver with a minimum of 1 secc
HLR-6	Functional	☐	ReviewedAndAccepted	SRATS-6	☐ Yes	☐ Yes	☐ No	☐ No	The CA subsystem software shall command the appropriate avoidance maneuver to the UAV autonomous flight control system
HLR-6.1	SafetyRelated	☐ Yes	ReviewedAndAccepted	SRATS-6	☐ Yes	☐ Yes	☐ No	☐ No	The UAS shall command an avoidance maneuver within 4 seconds of a collision avoidance maneuver being advised by the CAS.
HLR-8	Interface	☐	ReviewedAndAccepted	System architecture	☐ Yes	☐ Yes	☐ No	☐ No	The CA subsystem shall interact with the flight management system to provide commands for executing avoidance maneuvers
HLR-9	Interface	☐	ReviewedAndAccepted	System architecture	☐ Yes	☐ Yes	☐ No	☐ No	The CA subsystem shall retrieve sensor data
HLR-10	Interface	☐	ReviewedAndAccepted	System architecture	☐ Yes	☐ Yes	☐ No	☐ No	The CA subsystem shall interact with the guidance system to obtain information about the UAV's position, speed, altitude, and heading
HLR-11	Interface	☐	ReviewedAndAccepted	System architecture	☐ Yes	☐ Yes	☐ No	☐ No	The CA subsystem shall interact with the Ground Control Station, which serves as the command and control center for the UAV.

Figure B.16 HLRs and its DHLRs modelled with ReDaML - table view

- Detect traffic
  - Safety-critical component
    - Software component level : B
    - Partition: Partition Level B
  - Safety monitoring component
    - Software component level : B
  - Safety interface with << Track traffic >>
    - Interface between Software components
    - Interface for INTERNAL component
  - Safety Event: Sensor Fusion Issues (Negative effect on safety)
    - Error type: ServiceError
    - Kind: External
    - Description: Errors in fusing data from multiple sensors, leading to incorrect detection results or incomplete situational awareness.
    - Hazard(s) associated: Incorrect Collision Avoidance Decision, Inaccurate Traffic Information
  - Safety Event: Traffic detection time (Negative effect on safety)
    - Error type: TimingRelatedError
    - Kind: Internal
    - Description: Inadequate detection performance (without sufficient time) that can undermine the reliability and effectiveness of the collision avoidance system
    - Hazard(s) associated: Reduced System Reliability, Increased Collision Risk
- Track traffic
  - Safety-critical component
  - Safety interface with << Evaluate collision potential >>
  - Safety Event: Tracking Failure (Negative effect on safety)
  - Safety Event: Track Misidentification (Negative effect on safety)
  - Safety Event: Delayed traffic track (Negative effect on safety)
- Evaluate collision potential
  - Safety-critical component
  - Safety monitoring component
  - Safety interface with << Prioritize collision threats >>
  - Safety Event: Delayed Evaluation (Negative effect on safety)
  - Safety Event: Data Processing Errors (Negative effect on safety)
  - Safety Event: Incomplete Data Availability (Negative effect on safety)
- Prioritize collision threats
- Generate avoidance maneuver
- Command maneuver

Figure B.17 Safety-related HLRs modelled with ReDaML - tree view