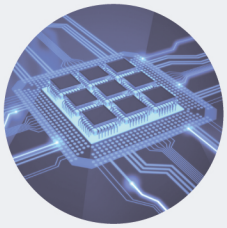
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA DE SISTEMAS COMPUTACIONAIS CRÍTICOS



Abordagem à especificação e verificação de requisitos: Caso de estudo em Updates via Over the Air

ANAISA CRISTINA PEREIRA CARVALHO
outubro de 2025

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

An approach towards rigorous specification and verification of requirements: Case study of Over the Air Updates

Anaísa Cristina Pereira Carvalho

Dissertation submitted in partial fulfilment of the requirements for the Master's degree in Critical Computing Systems Engineering

Supervisor: David Miguel Ramalho Pereira

Evaluation Committee:
President:
Luis Miguel Pinho, Instituto Superior de Engenharia do Porto

Members:
David Pereira, Vortex Colab
Jorge Coelho, Instituto Superior de Engenharia do Porto

Porto, September 30, 2025

Statement of Integrity

I hereby declare having conducted this academic work with integrity. I have not plagiarized or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

I declare that the work presented in this document is original and my own, and has not previously been used for any other purpose.

I further declare that I have fully followed the Code of Good Practices and Conduct of the Polytechnic Institute of Porto.

ISEP, Porto, September 30, 2025

Dedicatory

This thesis is dedicated to Fenix and Astrinha (my cats), whose constant presence and companionship were a source of comfort throughout this academic journey. Their curiosity of climbing to my keyboard and asking for attention also helped me breathing and relaxing a bit in times of higher stress, giving me a much needed pause. They were always an important presence, offering unconditional support without words.

To my friends that had the pleasure to live with me during this thesis process, a big thank you for always being there for me and still being my friends, I have no words to thank you enough for everything we passed together.

This thesis is also dedicated to every friend that never allowed me to give up on this academic achievement and it were always listening and giving me the encouragement and resilience to keep going, even with all the things life trough at me. Thank you for enduring my endless discussions and giving me a hug after.

To my family, especially my parents, who, despite not understanding the subject of this work, always encouraged me to give my best. I am deeply thankful for all the efforts you have made for me, which allowed me to come this far and reach this milestone. Without both of you, none of this would have been possible. I hope to repay, at least in part, the sacrifices you made so that I could go further, as you always gave your best for me.

With gratitude and affection, I thank everyone who accompanied me through this challenging journey.

Abstract

Modern vehicles rely on increasingly complex embedded software to deliver advanced features in performance, safety, and connectivity. In automotive software development, a precise requirement specification is essential to manage complexity and ensure reliability throughout the life cycle. Although natural language has been standard for documenting requirements, it often introduces ambiguity and misinterpretation. This thesis focuses on investigating the advantages of adopting formal requirements engineering over natural language approaches within the automotive field, using the Formal Requirements Elicitation Tool (FRET) tool and taking as use case a representative set of requirements of an Over-the-Air (OTA) updates project derived from the automotive industry. This analysis highlights how FRET improves clarity when eliciting, managing, analysis, and verifying requirements, thus serving as a strong asset for reducing errors in later development phases, which is specially important in critical projects. This work also provides evidence of the tangible benefits of formal techniques for the specification, analysis, and verification of vehicle software requirements, but also highlights that tools such as FRET still require formal logic training and, given the niche nature of the tool, that users maintain regular interaction with the developers in order to be aware of features that are fundamental for better usage of the tool, however that still remain undocumented in the literature.

Keywords: Over-The-Air, requirements engineering, formal requirements, natural requirements, FRET

Resumo

Os veículos modernos estão cada vez mais dependentes de uma elevada integração de software para oferecer recursos mais complexos relativamente ao desempenho, segurança e à conectividade do veículo. O desenvolvimento de software para a indústria automóvel requer que seja realizada uma especificação precisa dos requisitos, de forma a gerir a complexidade dos mesmos e garantindo uma elevada confiabilidade para toda a vida útil do veiculo. Embora o uso de linguagem natural seja o padrão para a escrita de requisitos, esta também introduz ambiguidade e leva a interpretações erradas dos mesmos. Esta tese investiga principalmente as vantagens de adoptar a utilização de uma linguaguem formal para a escrita de requisitios em deterimento do uso de uma linguaguem natural, focando esta aplicabilidade na industria automóvel. Este estudo tem como foco a adopção da ferramenta Formal Requirements Elicitation Tool (FRET) e o conjunto de requisitos em análise provem de um projeto de atualizações Over-the-Air (OTA) na industria automóvel. Esta análise pretende destacar como o FRET evidencia a clareza, gestão, análise e verificação requisitos desde a sua elicitação. Este recurso permite a redução de erros que apenas seriam encontranos em fases posteriores do desenvolvimento, e é esta mitigação que torna o uso deste tipo de ferramentas especialmente importante em projetos críticos. Este trabalho também apresenta evidências relativas aos beneficios da utilização de técnicas formais para a especificação, análise e verificação dos requisitos de software para veículos, mas também destaca que ferramentas como o FRET ainda exigem uma formação nos tópicos de lógica formal. Dado que o FRET ainda é uma ferramenta em desenvolvimento, é necessário que os utilizadores da mesma criem uma interação regular com os desenvolvedores, a fim de estarem cientes de recursos fundamentais para uma melhor utilização de todas as capacidades do FRET, que ainda não se encontram documentadas na literatura da mesma.

Acknowledgement

I would like to sincerely thank my supervisor David Pereira for the invaluable support, guidance, and encouragement throughout the entire process of developing this thesis. He was always present in the several iterations and improvements this work took, and helped me since the begining, on steering the direction of this research. He was also a big support, on not letting me drift from the main topic and the goal we defined. His expertise and patience were fundamental to the successful delivery of this work.

I also extend my deepest gratitude to Andreas Katis, one of the developers of the FRET. His assistance went well beyond expectations, providing detailed explanations on every Github issue and Discussion. He promptly addressed my doubts, and consistently delivered rapid and thorough support. I was genuinely impressed by the level of dedication and responsiveness, and without his support, my work wouldn't had reached such a deep level.

Contents

List of Figures

List of Tables

List of Acronyms

AUTOSAR	Automotive Open System Architecture.
CI/CD	Continuous Integration/Continuous Development.
CTL	Computation Tree Logic.
DoIP	Diagnostic over Internet Protocol.
EARS	Easy Approach to Requirements Syntax.
eCall	Emergency Call Systems.
ECUs	Electronic Control Units.
FRET	Formal Requirements Elicitation Tool.
GDPR	General Data Protection Regulation.
GPOS	General Purpose Operating System.
GUI	Graphical User Interface.
HSMs	Hardware Security Modules.
LTL	Linear Temporal Logic.
MBP	Model Based Projection.
MMUs	Memory Management Units.
NDA	Non-Disclosure Agreement.
NFRs	Non-Functional requirements.
OEMs	Original Equipment Manufacturers.
OTA	Over the Air.
QE	Quantifier Elimination.
RSLs	Requirements specification languages.
RTS	Real-Time System.
SDV	Software-Defined Vehicle.
SMT	Satisfiability Modulo Theories.
TLS	Transport Layer Security.

UDS	Unified Diagnostic Services.
UNECE	United Nations Economic Commission for Europe.

Chapter 1

Introduction

Requirements have a central role in software engineering since they define the objectives, functions, and constraints that shape the development of any system. Despite this importance, requirements in industry are often written in natural language, which introduces ambiguity, overlap, and inconsistency. These shortcomings are particularly critical in safety-sensitive domains such as automotive systems, where errors in early stages can propagate into failures with severe consequences.

With the above mentioned challenges posed by natural language requirements, and with software systems growing in complexity at a speed with no precedents, it becomes even more urgent the adoption of approaches to requirements elicitation, specification, and verification in software development processes.

This thesis investigates how requirements can be specified and verified in a more rigorous way by adopting an approach centered on the formal specification of requirements, supported by the Formal Requirements Elicitation Tool (FRET) [1] tool, developed in NASA by Ames Research Center [2], and that has being successfully used in the space domain. The work presented in this thesis aims also to explore the suitability of FRET in the automotive domain, by focusing on a small, yet relevant set of requirement of an Over the Air (OTA) update solution in a industrial setting, allowing to explore the challenges and potential benefits of structured approaches to requirements.

1.1 Context

The automotive industry has undergone a transformation driven by the increasing integration of software and electronic systems within vehicles [3, 4]. Modern vehicles rely on several Electronic Control Units (ECUs) to deliver complex features in areas such as safety, advanced driver assistance, connectivity, and infotainment [4]. As embedded system complexity grows, manufacturers must comply with a range of technical standards focused on safety, security, and functional reliability, with ISO 26262 serving as one of the most essential frameworks [5–7].

Meeting ISO 26262 requirements is critical not only for passenger safety and regulatory compliance but also for minimizing costly recalls arising from failures in software-controlled vehicle systems [6, 8]. The standard mandates systematic, traceable, and thorough requirements management across all abstraction layers, from vehicle level goals to software modules and components [8, 9]. These frameworks require traceability, formal documentation, and continuous improvement throughout the system lifecycle.

In parallel, the growing volume of in-vehicle software increases the need for effective management of updates and security patches over a vehicle's life. Over the Air (OTA) updates are a key approach to maintaining and enhancing vehicle functionality post-deployment. While OTA is not the main focus of this thesis, it provides the practical context and source of requirements analyzed here.

A fundamental challenge in this domain lies in the formulation of system requirements. In the automotive industry, requirements are still predominantly written in natural language [10–12]. This practice leads to widespread issues including ambiguity, duplication, and inconsistencies that undermine traceability, systematic verification, and change management as systems evolve [10, 12]. In safety-critical applications, these shortcomings are especially significant: errors originating from poor or unclear requirements can propagate through design and implementation, ultimately affecting both safety and security [13]. Recent studies confirm that ambiguity in requirements frequently leads to misunderstood or misapplied specifications, increasing unnecessary risks in to systems that demand high assurance [10, 11].

To address these challenges, researchers and engineers have explored more rigorous and structured specification methods, such as pattern-based, and computer aided formal specification approaches [14]. In this context, the Formal Requirements Elicitation Tool (FRET) is used in this thesis as the core tool for formalizing, analyzing, and verifying requirements in the automotive domain, while applied to an OTA update scenario. FRET has demonstrated value in handling safety-critical requirements in other domains and is here applied to automotive OTA requirements to evaluate its suitability and benefits in this field. This structured approach supports traceability and eases adaptation across design changes, aligning well with the iterative development cycles typical in automotive software [15, 16].

Accordingly, this thesis investigates how FRET can enhance the product development process for automotive systems by investigating how formal methods for requirements elicitation and verification improve the product development process for automotive systems. Using a real OTA project, the study evaluates the strengths and limitations of structured FRET-based specification relative to natural language practices, offering new insights for automotive requirements engineering.

1.2 Thesis Structure

This thesis is organized as follows: Chapter 1 gives an overview of the topic, its importance, the context, and the research methodology; Chapter 2 – State of the art, delves into the main topics and concepts that form the baseline on which the work of this thesis will be developed. It also includes a short critical analysis that is focused on industry standards that are relevant for the topics of the thesis; Chapter 3 introduces and justifies the main tools and frameworks used throughout our research, establishing the technical basis for subsequent analysis. Extended focus is given to FRET as it is the core tool that supported this work; Chapter 4 describes the OTA project that serves as a case study for this work and discusses the methodology and proposed approaches to requirements elicitation and specification. It also presents the main developments, analysis, and findings of this thesis. Finally, in Chapter 5, the main conclusions about the thesis work and results are discussed, as well as possible future research directions.

1.3 Privacy and ethics

This dissertation was developed using data provided by a company from the automotive field, solely for the purposes of understanding how the formal requirement elicitation of techniques can benefit this industry. To ensure confidentiality, a formal Nondisclosure Agreement (NDA) was established between Instituto Superior de Engenharia do Porto and the industry partner. All company-specific information has been anonymized, and every effort has been made to ensure that sensitive data cannot be associated with specific products, solutions, or clients. No personal data, human subjects, or survey results are included in this research. All project data was handled strictly for academic purposes and according to the university's privacy and ethics guidelines, and complies fully with the General Data Protection Regulation (GDPR). All sources and research articles used in this dissertation are properly credited to their authors.

1.4 Timeline

In the pre-thesis plan, the timeline presented was the one from Figure 1.1.

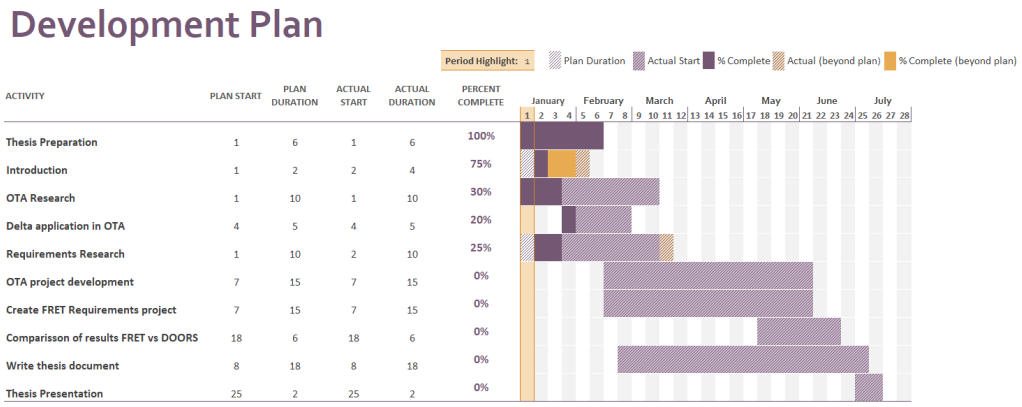


Figure 1.1: PreThesis development plan

However, this timeline was not executed as originally planned. The duration of each activity presented was extended in several weeks due to unforeseen personal and professional questions that lead to several inactivity periods during the course of development of this thesis making this plan not viable anymore.

From the activities represented, "Delta application in OTA" was removed due to the fact that the received requirements from the case study not contemplating this feature. The requirements research took a longer than predicted due to the learning curve on formal languages that the FRET tool uses. Regarding the planned weeks for the learning and use of the FRET tool, it was also affected by several FRET releases that occurred since this thesis topic was proposed until it was delivered. Each release had major changes and improvements that made sense the increased effort to present the newest possible FRET version and show their capabilities. Initially it was proposed to do a requirements comparison between FRET and DOORS, but during the development it made more sense that this comparison was made between FRET and EARS. In the pre-thesis development plan, it was also not accounted for the time that learning the FRET tool required due to the backend tools that exist and how they interact.

The thesis adopted a waterfall-inspired methodology to ensure scientific rigor and traceability throughout the automotive OTA requirements engineering case study. The process began with the analysis of the requirements provided in the case study. At this stage, substantial attention was dedicated to the redaction, selection and anonymization of requirements, ensuring that any sensitive or proprietary project information was carefully excluded from documentation.

Following the requirements cleanup, the next phase involved a comprehensive state-of-the-art review, focusing on requirements engineering practices in natural, semi-formal, and formal languages. This review established the scientific basis for comparing requirements standards and approaches relevant to the automotive context.

With the research finalized, the workflow progressed to the software requirements tools and a profound research regarding the FRET tool capabilities and limitations. It was also required to learn how the tools that allow FRET to make a complete analysis work, to make sure it aligned with the goals of this thesis. Then requirements were formalized using a semi-formal language and finally in the FRET tool, to illustrate and evaluate formal and natural language specifications, respectively. In this evaluation it was compared the EARS and FRET results.

Throughout all phases, the writing and documentation of the thesis was conducted in parallel with technical progress, milestone achievement, and validation activities. Although some technical tasks often required iterative refinement, the overall process and its presentation follow the clear, structured paradigm demanded by academic standards.

Chapter 2

State of the Art

Over the Air (OTA) software updates have become a central mechanism for delivering functionalities, security patches and performance improvements in connected systems. In the automotive sector, OTA enables continuous software maintenance while meeting demanding safety and security requirements. This domain therefore requires rigorous specification of functional and non functional requirements, particularly regarding security, safety and reliability [17, 18].

Requirements engineering provides the methodological foundation to specify, validate, and manage such requirements throughout the system lifecycle. Ambiguities and inconsistencies in requirements are a primary source of project delays, cost increase, and reduced system dependability [19, 20]. To mitigate these risks, several approaches have been proposed, ranging from natural language specifications to formal methods and dedicated management tools [21, 22].

This chapter reviews the state of the art in OTA update systems and in requirements engineering. First, it addresses current OTA paradigms, standards and typologies with emphasis on the automotive domain. Subsequently, it examines classification, specification and management of requirements, highlighting the techniques and tools that support their rigorous analysis.

2.1 Current paradigm in Over the Air (OTA)

The automotive industry is undergoing a digital transformation characterized by the emergence of the Software-Defined Vehicle (SDV) paradigm. Modern vehicles progressively integrate between 70 to 150 Electronic Control Units (ECUs) distributed across powertrain, safety, and infotainment subsystems. These control units collectively run over 100 million lines of code, a figure projected to surpass 200 million by 2030 as software-defined functions are increasingly incorporated [23–26]. The proliferation of embedded software intensifies maintenance demands and necessitates robust, scalable update mechanisms throughout the vehicle lifecycle.

Over the Air (OTA) update mechanisms have become essential, enabling remote deployment of software patches, feature enhancements, and security updates without requiring physical access to the vehicle. OTA technology has dramatically reduced logistical and financial burdens, with yearly global savings for manufacturers estimated at over 1.5 billion dollars due to digitalized recall resolutions [27, 28]. Additionally, consumer surveys indicate that up to 86% of users prefer digital updates rather than visits to authorized dealerships, evidencing strong market acceptance [27].

2.1.1 Regulations of Over the Air Updates in automotive

The concept of OTA updates germinated from earlier telematics and remote diagnostics systems. Automotive adoption of OTA was initiated in 2012 by Tesla, who pioneered the direct deployment of wireless firmware updates for both safety and performance improvements in the Model S platform. This milestone catalyzed widespread uptake and integration of OTA technology among leading Original Equipment Manufacturers (OEMs) such as Ford, General Motors, and Volkswagen, enabled by advances in cellular and Wi-Fi connectivity [16, 24, 27]. Initially limited to infotainment and non-critical functions, OTA now progressively supports safety and powertrain domains as architectures and cybersecurity mature. The European Union mandates integration of Emergency Call Systems (eCall) in new vehicles [3]. These safety-critical features rely heavily on software functions that must be securely and reliably maintained through validated updates [29].

Regulatory frameworks in Europe, including the General Data Protection Regulation (GDPR) and the United Nations Economic Commission for Europe WP.29 standards, increasingly shape market requirements for secure and compliant software maintenance. GDPR enforces transparency, user consent, and data protection, while UNECE Regulations No. 155 and 156 mandate comprehensive cybersecurity and structured software update management through robust OTA processes [30–32]. Compliance with standards such as ISO 26262 and SAE J3061 is now essential for functional safety and cybersecurity [29, 33].

OTA standards intersect data privacy regulations, automotive cybersecurity frameworks, and safety-critical software update management. Compliance with these standards is vital for enabling safe, secure, and efficient remote software maintenance of increasingly complex connected vehicles.

2.1.2 Challenges of Over the Air Integration

The complexity of vehicle electronic architectures increases the significance of robust OTA update needs. Continuous innovation, software version management, and real-time operational demands necessitate OTA solutions that maintain functional integrity and accommodate diverse hardware configurations [28, 34]. While communication protocols such as CAN, LIN, or Ethernet are standardized, OTA modules are not yet fully integrated into the Open Automotive Framework (AUTOSAR). The lack of universal OTA standardization, with over 30 proprietary solutions between OEMs and Tier-1 suppliers, leads to significant inefficiency and motivates standardization efforts by AUTOSAR and the eSync Alliance [13, 28]. The standardized approaches promise to reduce fragmentation, streamline development, and enable broader adoption between automotive brands.

2.1.3 Overview of Over the Air Architectures

Implementing secure, fault-tolerant OTA update capabilities requires microcontrollers equipped with specialized hardware features. OTA microcontrollers typically require extended embedded flash memory, to support advanced features such as Read-While-Write flash, memory management units (MMUs), dual-bank architectures, and dedicated cryptographic modules for secure key storage and authentication [35, 36]. This design significantly reduces system downtime during software switching, allowing the vehicle to continue normal operations while OTA updates are performed transparently in the background [25].

The OTA software utilizes one of several well-established update mechanisms adapted to the available memory architecture. Table 2.1 summarizes common OTA update methods based on hardware configurations:

Hardware	**Method**	**Method Explanation**
Internal memory only	In-Place method	The update software is downloaded and installed directly by overwriting the existing application stored in internal flash memory.
Internal memory + external flash memory	In-Place with backup method	The update is similarly installed directly into internal memory, but a backup copy of the previous software image is stored externally to enable rollback in case of update failure.
Internal memory partitioned into two banks	AB-Swap-Method	Two software images coexist in separate internal memory banks. One bank runs the active software while the other is inactive. The update is written to the inactive bank, then activated by switching memory banks after verification.
Internal memory + external flash with two external banks	A-B-A-Method	The currently running software resides in internal memory. The external flash memory contains two banks: one holds the backup software, and the other stores the incoming updated software. Upon successful verification, the updated software is copied internally for execution.

Table 2.1: Common OTA update methods distinguished by hardware memory configurations

The In-Place method involves directly overwriting the existing software stored in the internal flash memory with the updated version. When external flash memory is present, the In-Place with Backup method first saves a copy of the running software externally before proceeding with the overwrite, providing a recovery path if the update fails [37].

Both the AB-Swap and A-B-A' methods rely on a dual-bank memory architecture that isolates the active (memory A) and backup (memory B) software images to ensure robustness. In the AB-Swap method, one of the memory banks will be active - running the current software and the inactive memory bank will have the backup of the currently running software. When the new software is available, the OTA update will be performed to the inactive memory bank. When the update is finished it will perform the activation of the updated memory bank and switch the previous running one inactive making that software the backup one.[25]. The A-B-A' method is more complex. In the internal memory, we will have always the currently running software and in the external memory we will find the backup software (an exact copy of the currently running software) on one of the memory banks and the updated software will be written for the other memory bank. After the new software is downloaded to the external memory bank and all pre-conditions are checked it will be copied to the internal flash to be used. [38].

These differentiated OTA architectures address the trade-offs between resource constraints, update reliability, and fault tolerance, which are critical concerns in the automotive software update lifecycle [35].

2.1.4 Uptane framework for secure Over the Air Updates

Security is central to OTA deployment in automotive systems. Architectures commonly employ cryptographic signatures, secure boot processes, encrypted update packages, and hardware security modules to ensure the authenticity of updates and prevent unauthorized installations [13, 35]. Uptane is a dedicated security framework specifically designed to protect Over the Air (OTA) software update processes in modern vehicles from a broad range of cyber threats. Uptane addresses the unique challenges posed by automotive ecosystems, where numerous distributed electronic control units (ECUs) must receive secure and reliable software updates throughout a vehicle's lifespan [39–41].

Recognizing that some system components or cryptographic keys could be compromised during a vehicle's lifecycle, Uptane employs multiple layers of defense and a principle of separation of duties to ensure that an attacker must breach multiple independent control points to deliver malicious software or disrupt update operations successfully. This defense-in-depth approach significantly elevates the security baseline of OTA updates in automotive systems.

Uptane architecture segregates responsibilities between two critical repositories: the Image Repository (offline) and the Director Repository (online). The Image Repository houses digitally signed software images and corresponding metadata, rigorously ensuring that only authenticated and verified binary files are made available for deployment. Separately, the Director Repository maintains signed metadata that dictates which software versions are to be deployed to specific vehicles, orchestrating coordinated update campaigns across fleets. By decoupling these functions, Uptane enforces cost-effective security measures such as limiting exposure of offline, high-value signing keys, used by the Image Repository, from frequent online use, while enabling dynamic update management via online Director Repository keys [39–42].

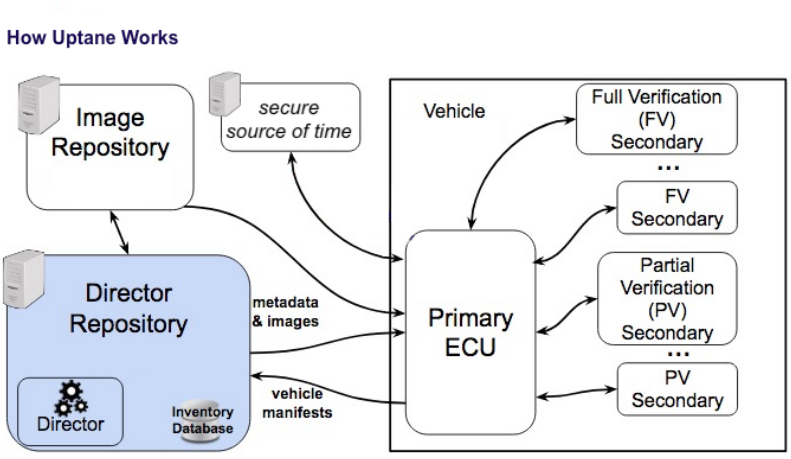


Figure 2.1: Uptane architecture extracted from [39]

The framework also supports advanced cryptographic features such as threshold signatures and hierarchical key management. This design simplification facilitates secure rotation, revocation, and recovery from compromised keys without inducing fleet-wide vulnerabilities. Uptane's cryptographic mechanisms comply with key industry cybersecurity standards, including ISO/SAE 21434 for automotive cybersecurity and the United Nations Economic

Commission for Europe (UNECE) WP.29 regulations (notably UN R155), which mandate robust cybersecurity risk management and secure software update practices for vehicle manufacturers [40, 42].

Updates are distributed over Transport Layer Security (TLS) to ensure confidentiality, authentication, and integrity of data in transit. TLS channels utilize manufacturer-managed digital certificates and private keys, often stored in hardware security modules (HSMs) or secure elements within backend infrastructure and sometimes vehicle hardware, to protect against man-in-the-middle and related network attacks [42, 43]. Platforms such as Airbiquity's OTAmatic (leading provider of automotive telematics solutions) have validated Uptane in large-scale deployments, enabling secure and differentiated software rollouts across OEM fleets [43, 44].

2.2 Requirement Specification

According to the Oxford English Dictionary, a requirement can be: "something that somebody needs or wants"; "something that is necessary according to a particular law or set of rules". Both descriptions can be applied to any system since requirements represent explicit statements of needs or conditions that a system must satisfy to fulfill its intended purpose. Requirements serve as the foundation upon which a project or system is developed, ensuring alignment with stakeholder goals and facilitating effective communication among development teams [19]. They define the necessary functionalities and qualities that a product must exhibit to meet user expectations and regulatory demands.

A well-constructed set of requirements functions as a blueprint for the development process, providing a clear understanding of system objectives and guiding the project toward successful completion [20, 21].

Requirements can be generally classified into two categories: functional and non-functional requirements. Functional requirements specify specific behaviors, functions, and interactions the system must support, detailing what the system should do. Non-functional requirements focus on system fulfillment. They address constraints on system operation, encompassing aspects such as performance, security, usability, reliability, and other quality attributes that define how the system operates within its environment [22].

Requirements can be documented in two different forms: using natural language or formal languages to write them. Natural language is the use of everyday language to describe what is needed. While this approach is straightforward and accessible, it can lead to ambiguities or misinterpretations between the people working on the project.

The first approach to mitigate these problems is enforcing some structure and suggesting requirement writing patterns, whose validation can be supported, up to some extent, by adequate tooling. However, this may not be sufficient to achieve the level of rigor that can be required, notably in safety-critical domains, so the direction is focusing on languages that have mathematical foundations, and that allow for well-principled and sound analysis and/or verification. Formal languages provide a more structured and precise way of writing requirements. This approach relies on mathematical or logical frameworks to specify requirements with precision and enable rigorous verification and validation [45]. The use of formal methods enhances the clarity and consistency of requirements, which is particularly critical in complex and safety-critical systems [46].

2.3 Requirement Classification

To classify requirements, it is important to understand the category where they belong. The most simple requirement specification is the division of requirements into functionality aspects: functional or non-functional requirements [19, 20, 47].

Functional requirements specify what a system or product should accomplish. They describe the essential functionalities and behaviors that the system must support to meet its objectives [21]. For example, a functional requirement might detail that a software application should allow users to register, log in, and perform transactions.

Non-functional requirements, on the other hand, focus on how the system or product should achieve its functional objectives. They address the quality attributes and constraints within which the system must operate, such as performance, security, and usability [22]. An example of a non-functional requirement could be that the application should respond to user requests within two seconds or maintain a certain level of security compliance.

2.3.1 Functional Requirements

A functional requirement typically defines the functionality of the product [19–21]. This represents all requirements that have a direct, measurable, and testable impact on the product. The requirements in this category define the behavior, capabilities, and characteristics of the product [22]. These kinds of requirements are useful for creating the baseline for product development and keeping the goals necessary to have a functional product [20]. These kinds of requirements are useful for creating the baseline for product development and keeping the goals necessary to have a functional product [20]. They can also be used as a link between the team responsible for the development of the product and the stakeholders [19, 20].

Some examples of topics that are provided in the functional requirements:

- Alert systems;
- Data storage;
- Data visualization;
- User authentication;

For instance, in the automotive sector, a functional requirement may specify that the vehicle must automatically notify emergency services with precise GPS location data within 30 seconds of airbag deployment in the event of an accident. This ensures timely emergency response and contributes directly to vehicle safety functionality.

2.3.2 Non-Functional Requirements

Non-functional requirements (NFRs) are all the requirements that describe the quality attributes and expectations of a product, such as usability, performance, reliability, security, scalability, and maintainability [19–22]. These requirements do not define what the system does, but rather how the system should perform its tasks and operate within its environment. NFRs set the guidelines and standards for the product's quality process, ensuring that the resulting project is robust, efficient, secure, and maintainable throughout its lifecycle.

NFRs are essential for ensuring that the system consistently exhibits the properties needed to meet stakeholder expectations and operational needs. They also act as overarching

constraints that span across functional areas, affecting the design, implementation, and future scalability of the system [20, 21]. Without proper attention to NFRs, the system might satisfy its basic functions but fail in terms of user satisfaction, adaptability, or long-term sustainability.

Some examples of topics that are provided in the non-functional requirements:

- Maintainability is responsible for monitoring the system and keeping logs of the behavior;
- Reliability requirements are responsible for the availability and recovery in case of fault errors;
- Scalability is responsible for handling the insertion of new features on the main project;
- Usability requirements respond to questions regarding the accessibility and user interface;
- Security: safeguarding the system from unauthorized access and ensuring data protection.

For example, in automotive systems, security-related non-functional requirements may mandate that OTA software updates must use strong authentication and encryption techniques to prevent unauthorized access or tampering. This protects the integrity of the vehicle's software and safeguards against potential cyber threats.

2.4 Requirement Languages

Requirements can be specified not only through different classification methodologies but also through diverse linguistic approaches. Depending on the desired level of precision and the intended audience, requirements may be expressed using either natural language or formal specification languages. Natural language is usually more suitable for high-level or stakeholder-facing requirements due to its accessibility, but it can introduce ambiguity and inconsistency. In contrast, formal languages offer mathematical rigour and are often preferred for technical or safety-critical specifications where precision is essential [19, 22, 47].

2.4.1 Natural language requirements

When writing requirements in natural language, the intention is to express project objectives and properties at a high level, making them easier to understand for all participants, including developers and stakeholders [19, 47]. However, natural language's inherent ambiguity and lack of formal structure often lead to misinterpretations, inconsistencies, and difficulties in verification and validation resulting in imprecisions due to multiple possible interpretations [20, 21].

Common causes of misinterpretation include ambiguity, vagueness, and lack of structural organization within the text. Such issues can lead to overly complex requirements documents, with missing requirements due to omissions, duplications due to different writing styles, and requirements wordiness that have long chains of thought that could be expressed in a simpler way [22]. Conflicting requirements may also arise, such as opposing conditions for system behavior, and textual analysis alone often makes detecting these conflicts difficult to detect [20].

Moreover, natural language requirements are challenging to verify prior to implementation since they cannot be directly simulated or rigorously tested, relying instead on subjective

interpretation [21]. To mitigate these issues, when writing natural language requirements, it is essential to focus on how the system must operate and interact with other systems (ensuring traceability), rather than solely on how the system should be built (implementation details) [48].

Despite these challenges, natural language remains the predominant form of requirements specification due to its simplicity and widespread comprehensibility. To improve precision and reduce ambiguity, complementary approaches like pattern languages use structured templates, combining natural language readability with greater rigor [47].

Failure in the interpretation and communication of automotive requirements can lead to profound consequences, including costly recalls, project delays, and safety risks. For instance, the 2022 recall of the Ford Mustang Mach-E electric SUV was principally driven by integration issues rooted in misaligned and incomplete requirements between hardware and software development teams. Similarly, Volkswagen's ID.3 faced significant launch delays due to inconsistent and ambiguous requirements across multiple suppliers, causing integration difficulties and rework. These real-world examples illustrate the critical importance of clear, precise, and unambiguous requirements in automotive software engineering to ensure functional safety, regulatory compliance, and overall project success [49, 50].

2.4.2 Pattern languages for specifying requirements

As acknowledged in the previous subsection, while natural language is widely used for writing requirements, this kind of approach has several problems related to interpretation and traceability of the requirements. However, directly using formal languages is often impractical because many stakeholders, including non-technical participants, require comprehensible specifications and knowledge of what is being described and required [19, 20]. To solve both problems it is necessary to create a middle ground between natural and formal languages of writing requirements. Pattern languages dedicated to writing and analyzing requirements offer such a compromise.

Several pattern languages exist that meet this criteria, including Simplified Technical English, Event-Condition-Action, and the Easy Approach to Requirements Syntax (EARS). Among these, EARS was selected for this thesis because of its simplicity and readability, making it accessible to a broad range of stakeholders [48].

EARS was designed to improve the clarity, precision, and consistency of requirements engineering. It addresses common challenges such as ambiguity and inconsistency in requirements documentation by providing a structured framework that facilitates the creation of well-defined requirements [48]. By utilizing a set of predefined templates that guide the formulation of requirements, it ensures standardization in language and format. This reduces the likelihood of misinterpretation, promoting better communication among different teams and fostering a common understanding [48].

Leveraging natural language, EARS remains accessible to stakeholders without technical backgrounds, ensuring inclusiveness throughout the requirements process. Additionally, EARS emphasizes contextual awareness, ensuring requirements are clear, relevant, and traceable from inception to implementation and testing. It is adaptable across domains and project scales, from small and straightforward initiatives to large, complex systems. This flexibility contributes to reliable outcomes and easier maintenance through consistent documentation [19, 48].

EARS is composed of 5 different templates that follow a specific syntax, presented in Table 2.2. For visualization purposes, a color code was created for the identification of all EARS elements.

- System name - Name of the system in study
- System Response - Described the response of the system
- Optional preconditions - Conditions that may precede an action
- Trigger - Event to initiate an action
- In a specific state - Special de-terminated state
- Feature is included - Feature that needs to be included

With the color code explained, in table 2.2 it is possible to see the correct classification and corresponding syntax for each EARS pattern.

Classification	**Syntax**
Ubiquitous	The SYSTEM NAME shall SYSTEM RESPONSE
Event-Driven	WHEN OPTIONAL PRECONDITIONS TRIGGER the SYSTEM NAME shall SYSTEM RESPONSE
State-Driven	WHILE IN A SPECIFIC STATE the SYSTEM NAME shall SYSTEM RESPONSE
Unwanted Behavior	IF OPTIONAL PRECONDITIONS TRIGGER, THEN the SYSTEM NAME shall SYSTEM RESPONSE
Optional Feature	WHERE FEATURE IS INCLUDED the SYSTEM NAME shall SYSTEM RESPONSE

Table 2.2: EARS structure syntax of requirements

These templates [48], are tailored to different types of requirements scenarios, ensuring all necessary elements are considered and explicitly stated:

- Ubiquitous - Requirement that has no preconditions or triggers;
- Event-Driven - Requirements that need to have a triggering event on the system in order to be initiated;
- State-Driven - Requirements that need to have the system in a determined state to be activated;
- Unwanted Behavior - Requirements related to undesired situations, or to mitigate the impact of those situations;
- Optional Feature - Requirements of optional system features;

In addition to these 5 specific EARS classifications, some requirements may not fit neatly into any of the standard categories. In such cases, it is possible to use a generic EARS structure by flexibly combining elements such as optional preconditions, triggers, and the system response as needed. A Generic template would look like this:

OPTIONAL PRECONDITIONS OPTIONAL TRIGGER the SYSTEM NAME shall SYSTEM RESPONSE

This adaptability allows EARS to accommodate a broader range of requirement types while preserving the benefits of clarity, consistency, and traceability inherent in the pattern-based approach [19, 48]. By extending the template in this way, engineers can ensure that less conventional requirements are still documented in a structured and comprehensible manner, supporting effective communication and future system validation.

The EARS classification will be used extensively in this thesis, particularly in Chapter 4, as an intermediary step between analyzing natural language requirements and formal specification. Examples from the EARS literature illustrate the practical application of different requirement patterns. For instance, an Event-Driven requirement such as "When 'mute' is selected, the system shall suppress all audio output" demonstrates how system behavior can be triggered by specific events [48, 51]. Similarly, the Optional Feature pattern is exemplified by the requirement: "Where the car has a sunroof, the car shall have a sunroof control panel on the driver door," highlighting conditional functionality dependent on system configuration [48]. These examples underline the clarity and structure that EARS brings to requirements engineering.

2.4.3 Formal language requirements

Formal languages are specifically designed to ensure that requirements are precise, unambiguous, and machine-readable. This level of rigor provides clarity and consistency, making such requirements particularly suitable for automated analysis and formal verification [19, 20]. Unlike natural language, which can often be vague or open to multiple interpretations, requirements specified using formal requirements specification languages (RSLs) employ structured templates and well-defined syntactic and semantic rules. This structure is crucial for preventing misunderstandings and supporting a shared understanding of system specifications [21, 22].

A key advantage of using formal languages is the ability to eliminate ambiguity, which is especially important in complex and safety-critical systems, where unclear requirements can lead to significant risks and costly errors. Examples of widely-used formal languages include Z, Linear Temporal Logic (LTL), Computation Tree Logic (CTL), each suited to different types of systems and domains [52].

By employing technical, rigorously structured notation, requirements expressed in formal languages become more easily verifiable, testable, and traceable. This facilitates automated checking for consistency, completeness, and compliance with desired properties, reducing errors introduced during the software development lifecycle. As a result, formal specification languages support higher confidence in system correctness and safety.

While formal languages provide these benefits, they also present challenges in terms of accessibility and required expertise from stakeholders without formal methods training may find them less intuitive than natural or pattern-based languages. Nevertheless, in domains demanding high assurance, the adoption of formal requirements is a recognized best practice for managing and verifying critical project properties [19, 20].

Two prominent examples highlight the practical impact of formal requirements in the automotive domain. Toyota extensively applied formal methods for its brake-by-wire system,

leveraging formal specification languages and model checking to rigorously define and verify safety-critical timing and functional properties [19, 20]. This practice was instrumental in ensuring compliance with ISO 26262 and in preventing design errors that could compromise braking safety. Similarly, Volvo integrated formal languages such as the B-Method in developing autonomous vehicle control systems, enabling precise specification of collision avoidance and lane-keeping algorithms [21, 22]. The use of formal requirements facilitated exhaustive verification and validation, substantially enhancing the reliability and safety of complex automotive control functions.

2.5 Requirement Management and Analysis Tools

Requirements form the backbone of any development project, and as projects evolve, requirements often increase in number, complexity, and interdependence. Effective management of these requirements is vital to maintain traceability, manage changes, ensure consistency, and support collaboration among diverse stakeholders [21]. Without adequate tools and methodologies, manually handling such growing and evolving requirements can lead to omissions, inconsistencies, and miscommunication, adversely impacting the quality and success of the project.

Formal specification notations complement these tools by enabling automated verification and validation, thereby streamlining quality assurance processes and reducing errors introduced during system development [53]. The integration of formal languages with natural language requirements management facilitates bridging the gap between stakeholder understanding and technical rigor.

For the purposes of this thesis, three key tools have been selected to support the complete requirements lifecycle, each addressing specific needs and stages:

- **IBM Rational DOORS** — An industry-leading enterprise requirements management system widely adopted in the automotive domain. It supports complex project demands including version control, traceability, impact analysis, and stakeholder collaboration. It was selected for this study due to its established use in the case study project [54].
- **DOORSTOP** — An open-source, lightweight tool designed for authoring and managing requirements primarily in natural language. DOORSTOP provides flexibility with repository-based version control and simplicity for projects that do not require the overhead of large enterprise systems [55].
- **FRET** — Developed at NASA Ames Research Center, FRET offers formal requirements analysis leveraging structured specifications and automated verification. It enables identification of inconsistencies, ambiguities, and incompleteness, bridging the gap between natural language requirements and formal methods [53].

These tools cover diverse requirements engineering activities, from initial stakeholder communication to rigorous formal verification. Their complementary features allow for a flexible yet robust management approach, critical for complex, safety-critical automotive projects.

The following chapter 3 provides a comprehensive and comparative analysis of these tools within the context of this thesis, detailing their capabilities, integration workflows, advantages, and limitations.

Chapter 3

Requirement Analysis Tools

Requirements analysis is a complex task when the initial requirements exist as lengthy, unstructured lists contributed by various stakeholders involved in a project. To overcome this challenge and enable systematic analysis, a variety of management tools and methodological approaches have been developed to facilitate the transformation and organization of requirements for better analysis. This chapter presents a focused examination of selected tools that support requirements transformation and analysis through different approaches. Specifically, it reviews **IBM Rational DOORS**, a widely recognized industry-standard tool for comprehensive requirements management [56]; **Doorstop**, an open-source solution that provides flexible and collaborative requirements tracking; and **FRET**, an open-source tool that enables formal requirements specification based on a rigorous pattern language rather than natural language [57]. These tools were selected to illustrate a spectrum ranging from established industrial practice to open-source flexibility and formal rigor.

3.1 IBM Rational DOORS

IBM Rational DOORS (Dynamic Object-Oriented Requirements System) is a widely adopted requirements management tool that primarily utilizes a natural language approach for specifying requirements. It enables the tracing, analysis, and management of system requirements throughout all phases of the project life cycle. This software is employed in critical industries such as defense, healthcare, and automotive industries, where the management of complex systems and compliance with rigorous standards are essential [56].

At its core, DOORS organizes requirements within a robust, database-driven framework designed to support large-scale projects. The following key capabilities facilitate effective requirements management:

- **Collaboration and Management:** DOORS supports a centralized server environment where all project stakeholders have controlled access to a shared repository containing requirements from all teams. Each team designates responsible individuals who oversee modification tracking and review processes, thereby fostering alignment and collaboration across the project.
- **Requirements representation:** Each team is responsible for their set of requirements. Teams structure their requirements into hierarchical trees aligned with component or system architectures. Such organization simplifies the management of extensive requirement sets and clarifies the relationships between them.
- **Traceability:** Requirements can be linked to represent dependencies within and across systems. Additionally, DOORS enables association of requirements with related project

artifacts such as system documentation, detailed designs, and test cases, ensuring end-to-end traceability.

- **Versioning control:** Tracking changes is one of the biggest advantages of using a requirement management tool. The tool implements a version control mechanism that tracks all changes to requirements. Proposed modifications undergo review and approval by designated personnel, maintaining the integrity and consistency of the requirements baseline.
- **Teamwork:** Multiple users can work on the same project simultaneously, with the changes tracked and documented. DOORS allows to have the project divided into several sections, and they have a lock mechanism for each section. They have a system of permissions that defines who can modify each section. Sections that allow comments, discussion threads, and formal review processes also foster an environment of collaboration.
- **Multi text support:** DOORS accommodates diverse requirement content types, including textual descriptions, tables, images, and hyperlinks, enabling comprehensive documentation within requirement entries.
- **Management and Reporting:** Each requirement has a status (new, in progress, completed, obsolete). Those statuses can be updated according to the project development respecting the life-cycle that is defined.
- **Filters:** Users can create personalized filters and views based on criteria such as status, priority, or responsible team members. These filters reduce information overload and focus attention on relevant requirements subsets.
- **Scalability:** DOORS is designed to maintain performance and usability when managing thousands of requirements, making it suitable for complex, large-scale projects [19, 56].

Considering all the capabilities mentioned above, IBM Rational DOORS helps teams organize, trace, and manage their requirements throughout the life of the project. The objective is to have all requirements written consistently and ensure that they are all implemented. DOORS is particularly suited for large and complex projects where traceability, compliance, and change management are critical. Its capabilities for customization, integration with other tools, and support for collaboration have contributed to its widespread adoption in regulated industries.

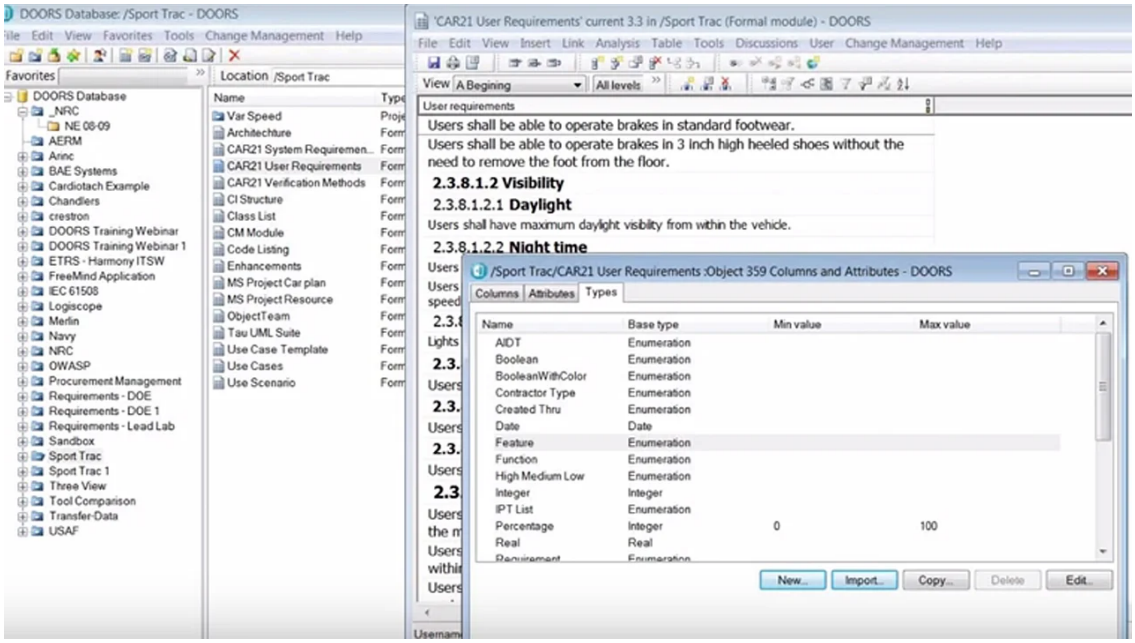


Figure 3.1: IBM Rational DOORS Graphical Interface

Despite these advantages, Rational DOORS has certain limitations, including a user interface that some users find less intuitive, potential performance inefficiencies with very large datasets, and challenges related to customization and integration with third-party tools. IBM has addressed many of these issues in DOORS Next Generation, a more modern platform intended to eventually replace the classic DOORS [56]. While Rational DOORS remains a leading tool in the field, alternative solutions such as the open-source tool Doorstop exist and may be considered depending on project context and requirements.

3.2 DOORSTOP

Doorstop is an open-source requirements management tool that stores requirements as text entries in YAML files. Unlike some tools that perform internal semantic analysis, Doorstop focuses on flexible and lightweight management without analyzing the requirements themselves [58]. This design choice enables adaptability for diverse projects and integration with existing development workflows.

Main advantages of Doorstop:

- **Project's version Control:** Requirements can co-exist within the same code repository as software artifacts, allowing the use of version control systems such as Git to track and manage changes effectively.
- **Hierarchical structure:** Requirements are structured in a hierarchical tree format, facilitating the handling of complex projects. Additionally, documents can link to other documents, providing a comprehensive documentation system.
- **Customizations:** Each requirement supports customizable attributes such as status, priority, or other user-defined properties relevant to the project context.
- **Traceability:** Doorstop incorporates integrated mechanisms to validate requirement traceability and ensure all links and dependencies are properly maintained.

- **Reporting:** The tool generates various visual representations of the requirements data in HTML format, simplifying sharing and review among stakeholders.
- **Cross-Platform:** Doorstop is available across the major operating systems (Linux, Windows, and macOS). It is lightweight and portable, which supports seamless adoption.
- **Automation:** Doorstop customizable nature allows integration with Continuous Integration/Continuous Development (CI/CD) pipelines, for example through Jenkins. This enables automated compliance and quality checks during software builds.

Doorstop's flexibility and open-source availability are among its greatest strengths, making it a compelling choice for small to medium-scale projects and for organizations seeking cost-effective requirements management solutions. To improve user interaction, a cross-platform API known as doorstop-edit provides a Graphical User Interface (GUI) [59].

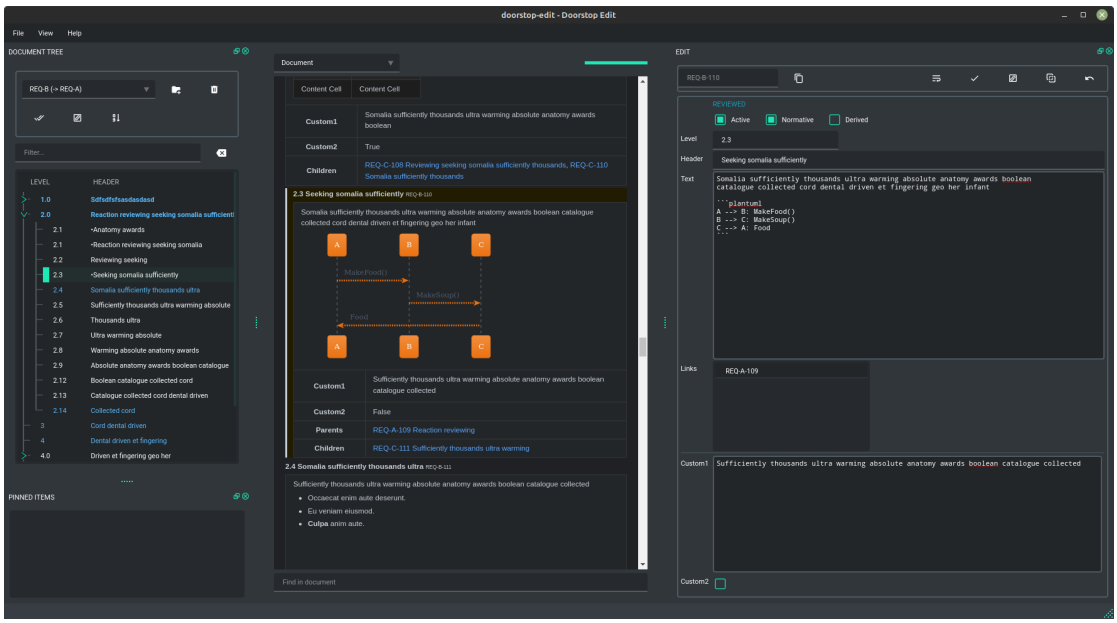


Figure 3.2: Doorstop GUI

Normally, Doorstop is most commonly found in small-to-medium projects and in industries where maintaining requirement traceability is essential. Although not being as prevalent as IBM Doors, there are several examples of relevant projects that adopted Doorstop:

- **Space ROS (NASA / Open Robotics):** Space ROS integrates Doorstop into its certifiable development workflow, using it to capture and review requirements within Git repositories and to generate traceability reports. The process is aligned with NASA's Formal Requirements Elicitation Tool (FRET), ensuring consistency between high-level mission requirements and software verification artifacts.
- **Zephyr Real-Time Operating System:** The Zephyr project has explored Doorstop to meet safety-related standards such as IEC 61508. Community discussions and prototype repositories document how functional and safety requirements can be expressed in Doorstop and linked to tests and code elements (e.g., through Doxygen tags).

- **Xen Project – Functional Safety Special Interest Group:** Within the Xen hypervisor ecosystem, the Functional Safety SIG employs Doorstop to maintain and publish structured requirement sets. Their publicly available "Development Process and Traceability" material illustrates end-to-end requirement management, including change-impact analysis and generation of formal compliance reports.
- **ROS Safety Working Group – Requirements Playground:** The ROS Safety Working Group provides an open demonstration repository, requirements playground, which showcases a complete workflow for capturing system and software requirements in YAML via Doorstop and publishing traceability information in HTML format.

However, and despite its clear relevance for the requirements engineering community, we will not further explore it in this thesis, as it lacks the support for formal requirements' specification.

3.3 Formal Requirements Elicitation Tool

The Formal Requirements Elicitation Tool (FRET) is an open-source software developed by NASA to facilitate the elicitation and formal specification of system requirements [60].

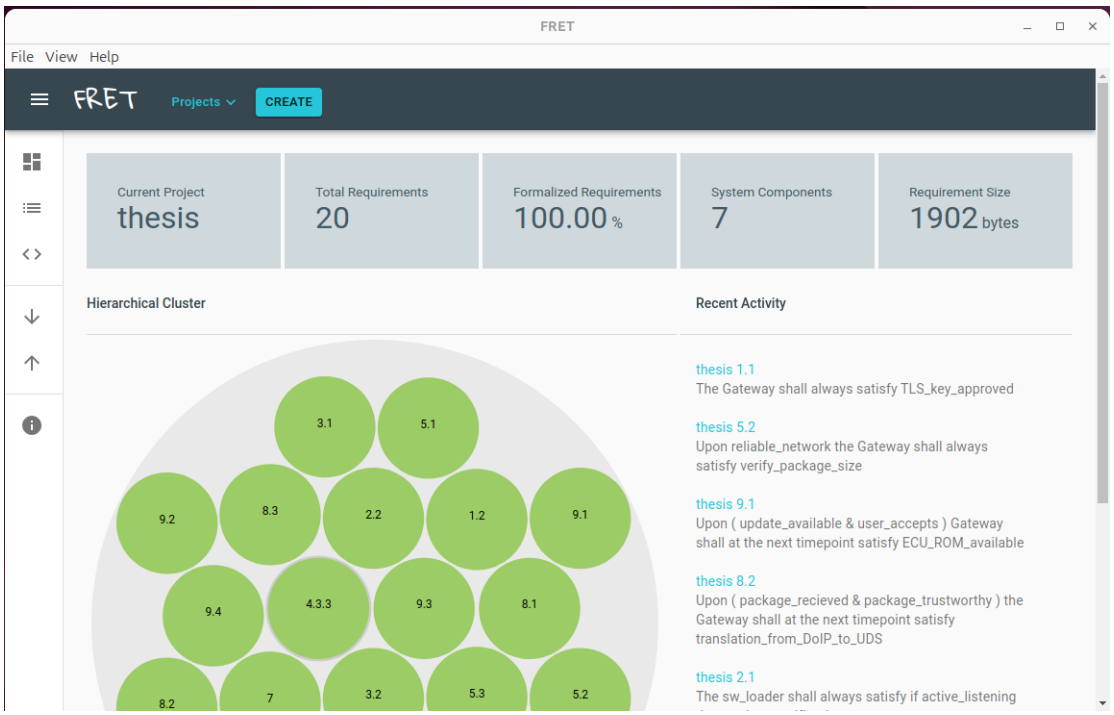


Figure 3.3: FRET initialization after project creation

FRET is designed to enhance the requirements engineering process by employing formal methods that enable precise, unambiguous, and verifiable requirements specifications. FRET supports the rigorous analysis and verification of requirements, addressing challenges commonly encountered in safety-critical and complex system development.

- **Flexibility:** FRET provides customized views and filters that can be shared among developers, allowing stakeholders to tailor their perspective according to project needs.

- **Formalization:** Requirements are expressed using a patterned language that is automatically translated into formal specification languages. This mathematical formalization enables rigorous specification, verification, and validation of system requirements.
- **Standardization:** By enforcing a rigid and deterministic syntax for requirement statements, FRET reduces ambiguity and inconsistency, thereby enhancing precision and reducing errors and inconsistencies.
- **Validation:** The tool includes verification mechanisms to check whether requirements have been correctly implemented and validation procedures to assess if the requirements meet their intended purposes. These capabilities help identify issues early in the development lifecycle.
- **Improved Communication:** The use of a standardized formal language facilitates clearer communication among stakeholders, minimizing misunderstandings and misinterpretations.
- **Comprehensive Analysis:** Through integration with model checkers and simulators, FRET supports rigorous analysis of requirements to ensure they are complete, consistent, and feasible.
- **Simulation Interactive:** Users can simulate requirement behavior, including temporal aspects, to analyze how they perform in practical scenarios.
- **Integration:** FRET interfaces with a range of model checkers and simulation tools to further automate the analysis and verification process.

Unlike IBM Rational DOORS, which provides flexibility to write requirements in natural language, FRET requires adherence to specific, structured patterns. While this restricts free-form requirement writing, it is essential for enabling the tool's formal verification and simulation capabilities, a trade-off well suited for high-assurance system development.

This chapter introduces FRET's fundamental concepts and capabilities. It is valid for the version 3.0 of FRET and serves as the foundation for detailed exploration of its analysis techniques, supported verification and simulation tools, variable mapping approaches, and its specialized specification language, Fretish, in the following sections.

3.3.1 The Fretish Language

To support rigorous yet accessible requirements specification within the FRET tool, a domain-specific language named *Fretish* was developed. Fretish is designed to balance ease of use by resembling natural language in style and syntax, with well-defined formal semantics that enable unambiguous interpretation and formal analysis. This design facilitates collaboration among diverse stakeholders, from system engineers to managers, without requiring expert knowledge of formal methods.

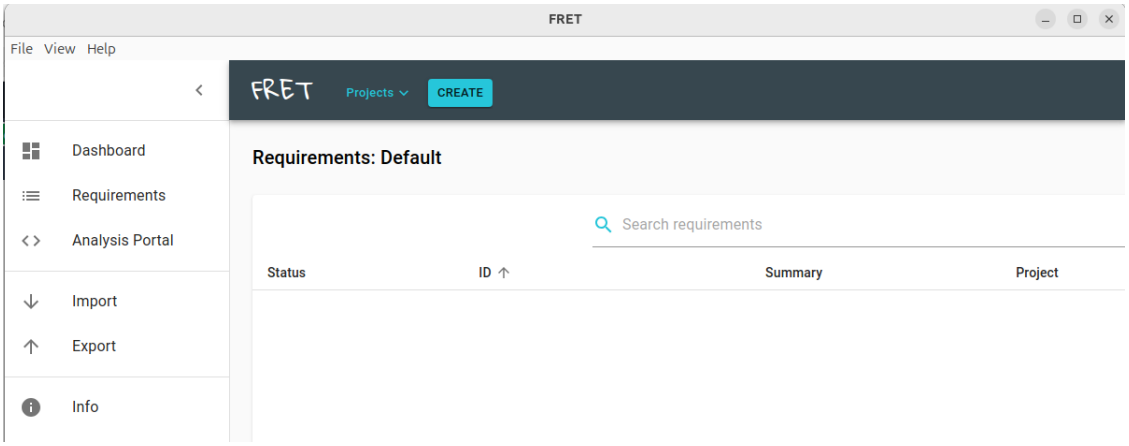


Figure 3.4: FRET requirements creation

Fretish employs a template-based approach, guiding users to construct requirements conforming to common specification patterns such as state assertions, timing constraints, and conditional behaviors. These templates enforce a structured format that enhances clarity and enables automated translation into formal verification languages.

Requirement structures

Every Fretish requirement is composed of several elements, and it exists two modes for requirement writing. The classical FRET mode is composed by five elements and is shown in Figure 3.5). The probabilist mode is composed by six elements, and is illustrated in Figure 3.6):

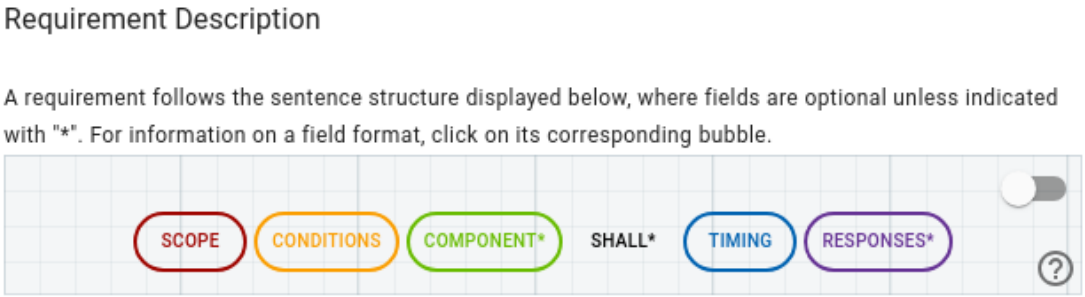


Figure 3.5: Requirement description mode of classical FRET

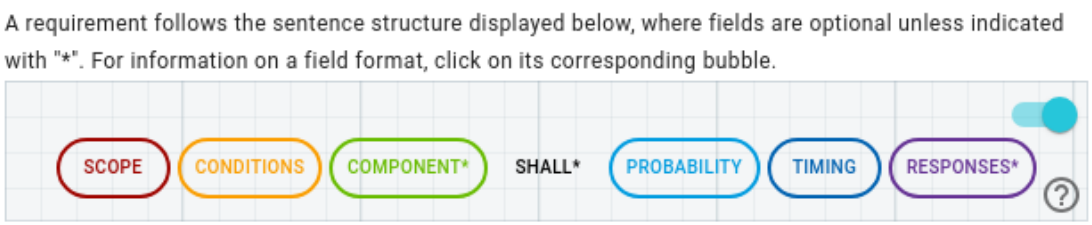


Figure 3.6: Requirement description mode of probabilist FRET

The selection and switch between the two modes is available through the toggle button, that when active it activates the probabilistic FRET mode, shown in figure 3.6.

- **SCOPE**: Defines the interval or mode during which the requirement applies. Scope is optional but, if given, restricts the requirement evaluation to system behavior within a specified
- **CONDITION**: Describes the circumstances under which the response is triggered. Conditions can be of two types:
  - *Triggering*: The requirement becomes active upon a condition becoming true.
  - *Continual*: The response must hold continuously as long as the condition remains true.

  Conditions may be complex Boolean expressions joined by logical operators `and`, `or`, and negation.
- **COMPONENT:** Specifies the component of the system to which the requirement applies.
- **TIMING:** Defines temporal constraints on when the response must occur relative to the condition and scope.
- **RESPONSE:** Specifies the expected behavior or action the component must perform to satisfy the requirement.
- **PROBABILITY:** Specifies the probability assigned to the timed response. It can be empty if the requirement is not probabilistic.

Logical expressions and operators are fundamental for formally specifying requirements. They enable precise definition of system conditions by combining predicates through logical connectives such as conjunction, disjunction, and implication. This formalism eliminates ambiguities inherent in natural language, supports automated consistency checks, and facilitates formal verification processes.

Timing constraints are crucial for specifying temporal aspects of requirements, especially in real-time contexts. FRET uses temporal operators to express constraints like eventuality, bounded delays, and ordering of events, which ensure correct timing behavior and system responsiveness. These constraints prevent under specification and enable rigorous temporal verification and runtime monitoring.

3.3.2 FRETish Templates

FRET incorporates a set of integrated templates designed to simplify and standardize the process of requirements specification. These templates are also called the FRETish language. FRETish guide users in expressing different types of system requirements using syntax that is both human-readable and suitable for formal analysis [61]. By following FRETish patterns, users can describe requirements consistently, helping prevent ambiguity and improving the analyzability of specifications.

Each Classic FRETish patterns is structured around five key elements: Scope, Condition, Component, Timing, and Response, which are highlighted using color coding for clarity in both visual displays and tabular representations. This organization allows users to quickly identify and compose each part of a requirement:

- Scope - The operational mode or interval in which the requirement must hold.
- Conditions - The circumstances or triggers that activate the requirement.
- Component - The system module to which the requirement applies.
- Probability - The associated probability of the requirement.
- Timing - Temporal constraints that specify when a response must occur relative to the condition.
- Responses - The expected outcome or action that the component should fulfill.

With the color code explained, in table 3.1 it is possible to see the correct classification and corresponding syntax for each FRET template. It is worth mentioning, that the probability element was only added in FRET version 3.0, but this addition did not changed the structure of the already existing in the tool, nor new templates were introduced. This element is only used when the user decides to manually integrate it in the requirement writing and it can be by user modification of one of the following templates.

Table 3.1 show the FRETish templates, their names, and their representative syntaxes, with color codes annotating each key element. Illustrated figures accompany each template.

Figure	**Name**	**Syntax**
3.8	Change State	<component> shall always satisfy if (<input state> & <condition>) then <output state>
3.9	Process Command	Upon <command> the <component> shall <timing> satisfy <response>
3.10	Check Bounds	The <component> shall always satisfy <bounds>
3.11	Set Diagnostic Flag	<condition> the <component> shall <timing> satisfy <response>
3.12	Prescribe Format	The <component> shall always satisfy <response>
3.13	State Transition	Upon (<input state> & <condition>) <component> shall at the next timepoint satisfy <output state>
3.14	State Transition Stay Pre	<component> shall always satisfy if preBool(false, <state> & !(<outgoing guard disjunction>)) then <same_state>

Table 3.1: FRETish templates

Upon FRET installation, we have in the assistance tab the above mentioned 7 templates, as shown in Figure 3.7. However, the FRET tool allows the user to create their own templates by using the templates editor with the glossary. This allows FRET to be more adaptable to the projects.

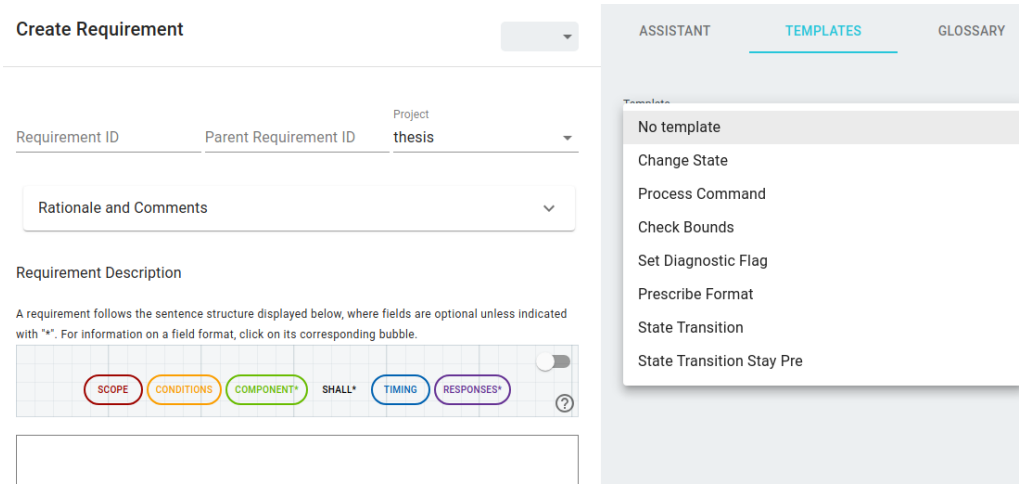


Figure 3.7: All available templates

Change State:

This template is used to describe how the state of a system component changes. It specifies the input state, additional conditions, and the required output state. The relationship between input and output states is defined in a pre-post relationship. It is accessible in FRET's user interface as presented in Figure 3.8.

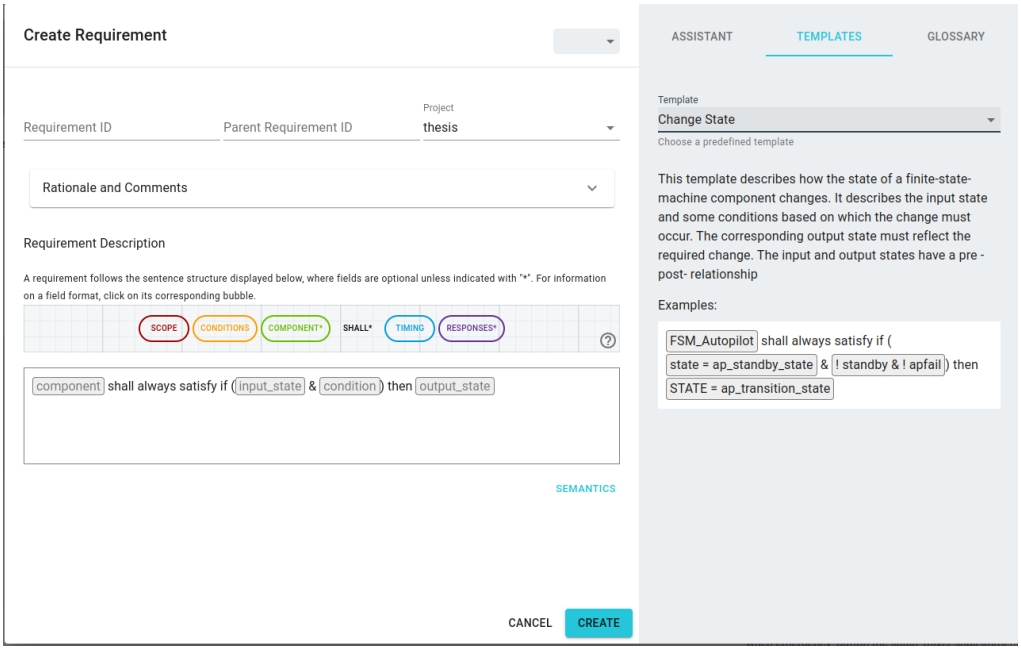


Figure 3.8: FRETish template: Change State

Process Command:

This pattern specifies how a command should be processed, detailing the component, timing requirements (e.g., immediate or delayed response), and the expected system response. It is accessible in FRET's user interface as presented in Figure 3.9.

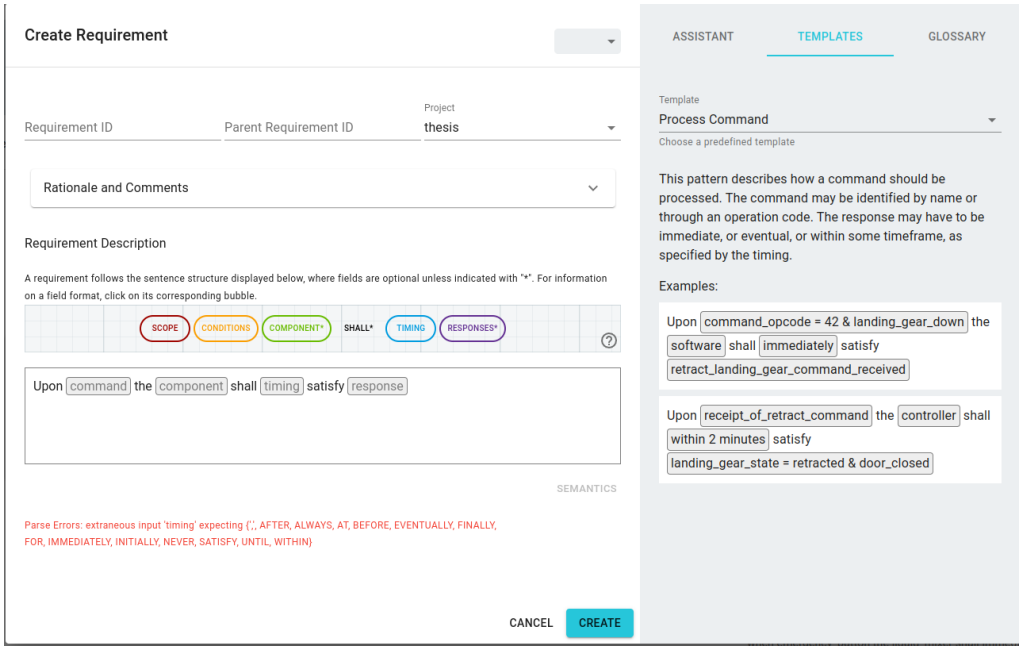


Figure 3.9: FRETish template: Process Command

Check Bounds:

This template expresses requirements ensuring that a signal or parameter remains within prescribed bounds (e.g., temperature not exceeding a threshold). It is accessible in FRET's user interface as presented in Figure 3.10.

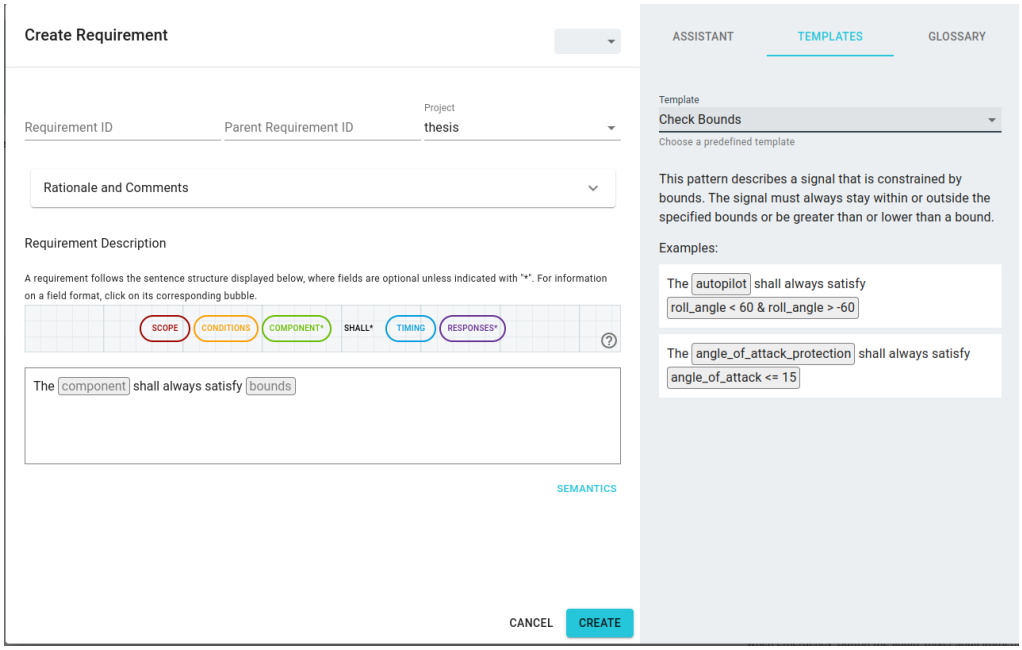


Figure 3.10: FRETish template: Check Bounds

Set Diagnostic Flag:

Used to specify that the system must set a diagnostic flag when certain conditions occur. The response is typically expected to be immediate, but other timing operators may also be used.It is accessible in FRET's user interface as presented in Figure 3.11.

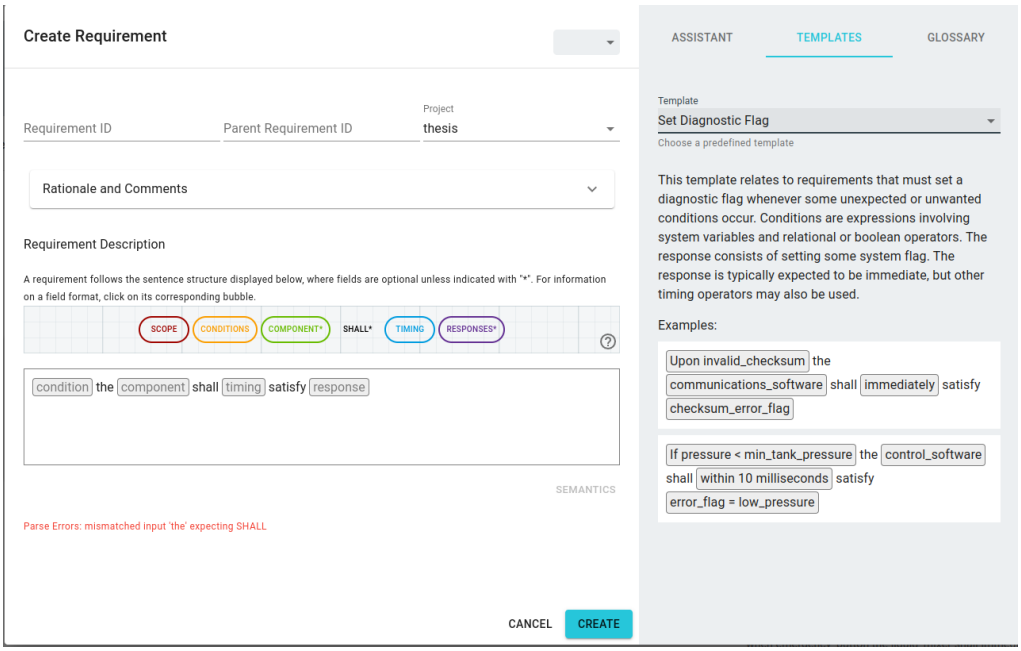


Figure 3.11: FRETish template: Set Diagnostic Flag

Prescribe Format:

This template defines the format for commands, telemetry messages, or other standardized data structures according to a referenced specification or standard. It is accessible in FRET's user interface as presented in Figure 3.12.

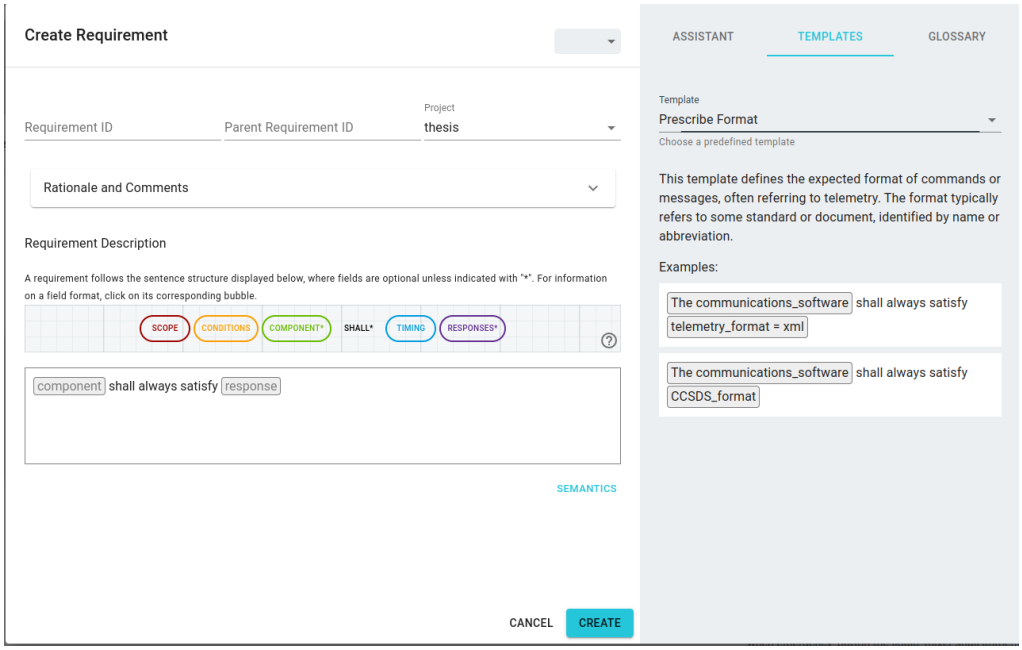


Figure 3.12: FRETish template: Prescribe Format

State Transition:

Used to formalize transitions within state-machines, describing the triggering condition and resulting state change at the next timepoint.It is accessible in FRET's user interface as presented in Figure 3.13.

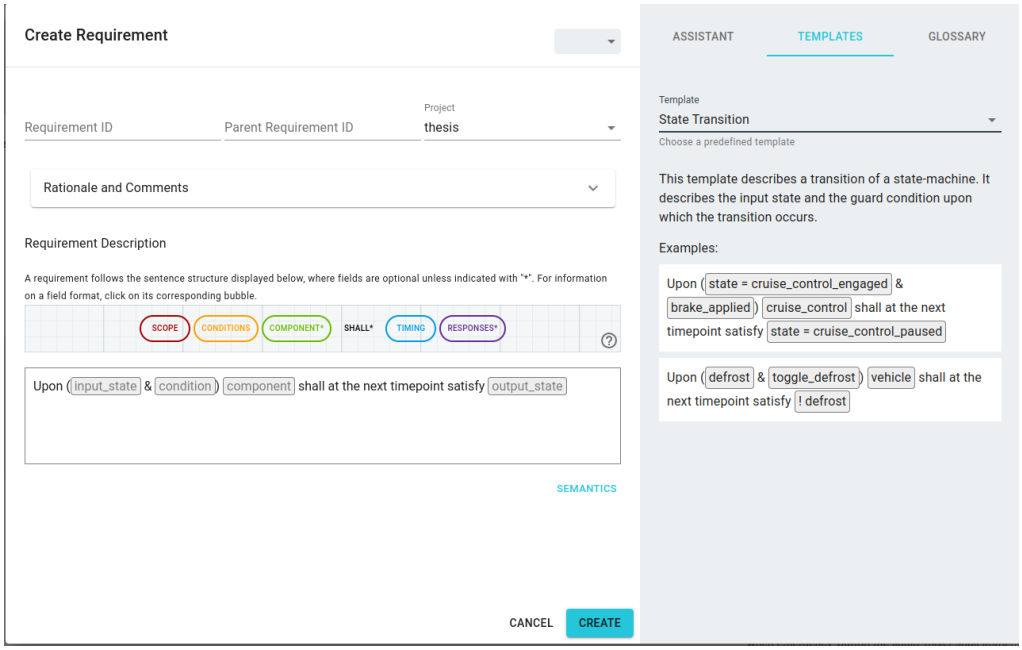


Figure 3.13: FRETish template: State Transition

State Transition Stay Pre:

This template formalizes the requirement for a state-machine component to remain in its current state, based on negation of possible outgoing transition conditions.It is accessible in FRET's user interface as presented in Figure 3.14.

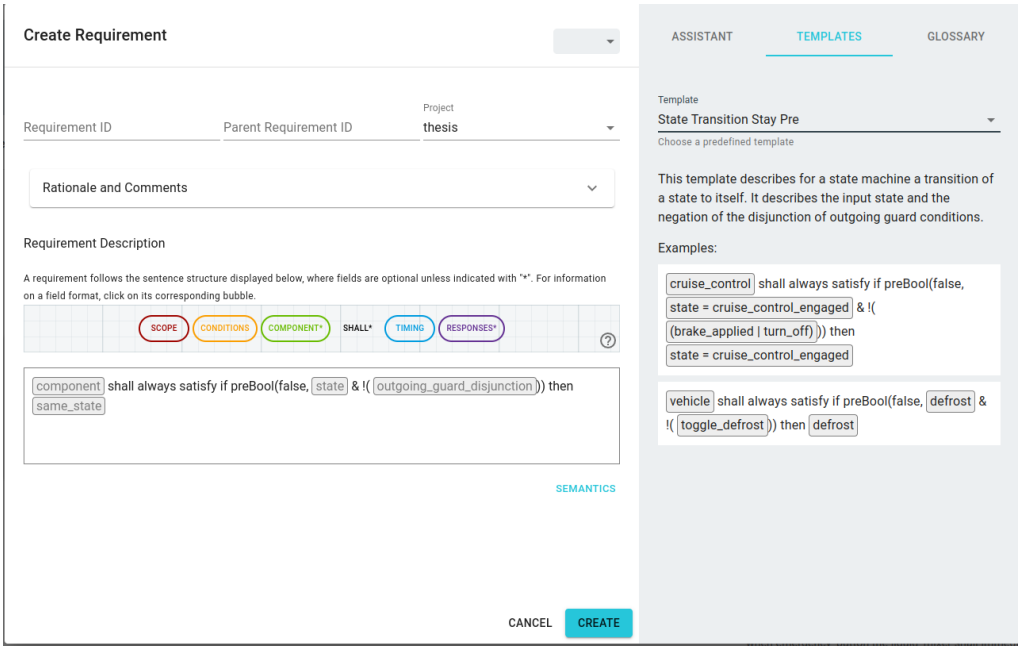


Figure 3.14: FRETish template: State Transition Stay Pre

Probability element:

Since the probability element was recently added to FRET, the user can use a base template from the ones presented above and just insert the probability associated to the requirement by toggling this feature, or it can write a requirement without a specific template. It is accessible in FRET's user interface as presented in Figure 3.15.

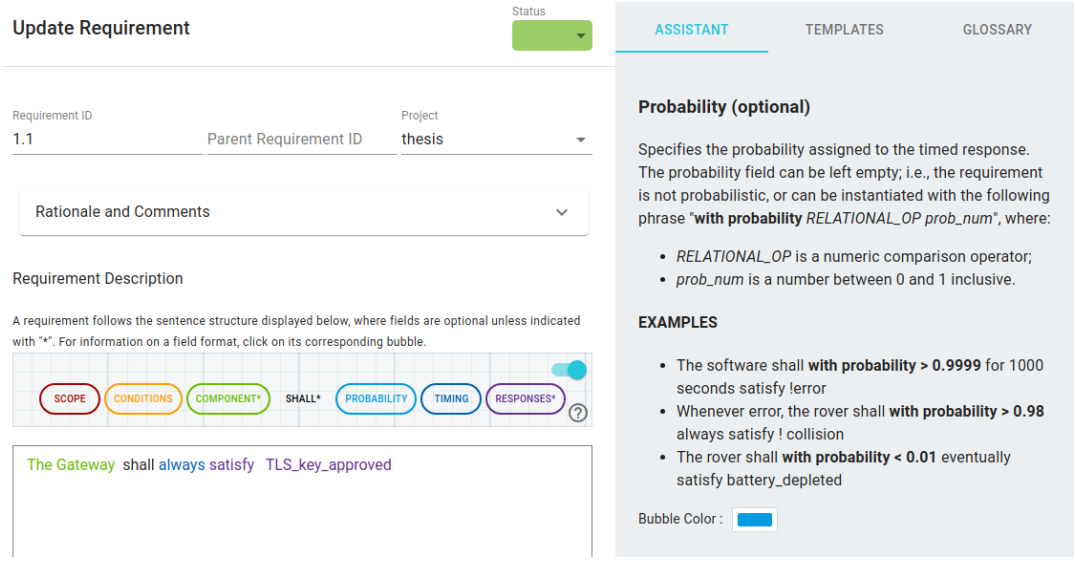


Figure 3.15: FRETish template: Probability mode active

In addition to the presented built-in requirements templates, the written documentation of the tool mention some more general templates that can be of highly relevance in safety-critical projects. Examples are:

- **General Temporal Requirement:** "In <scope>, if <condition> holds, then <action> must happen within <timing>."
- **Safety Requirement:** "The system shall always satisfy <condition>."

3.3.3 Variable mapping

The user, writes the requirements in FRETish. To allow any of integrated tools to understand the requirements themselves, is necessary to translate them into mathematical components. This is done in the Variable Mapping tab of the FRET analysis tool. The figure 3.16 shows all the possibilities to do this attributions to the components from the FRET templates. When the goal is to use an external tool like Ogma, those can be just direct exported to CoPilot and the user is not obligated need to map the variables in FRET itself. To use the FRET abilities is mandatory for the user to map the variables of the written requirements. This map can be in CoCoSim to allow the Realizability Checking and Time simulations, or it can be um SMV if the goal is just to Generate Test Cases for the requirements. All these FRET functionalities are described in detail in their following sections.

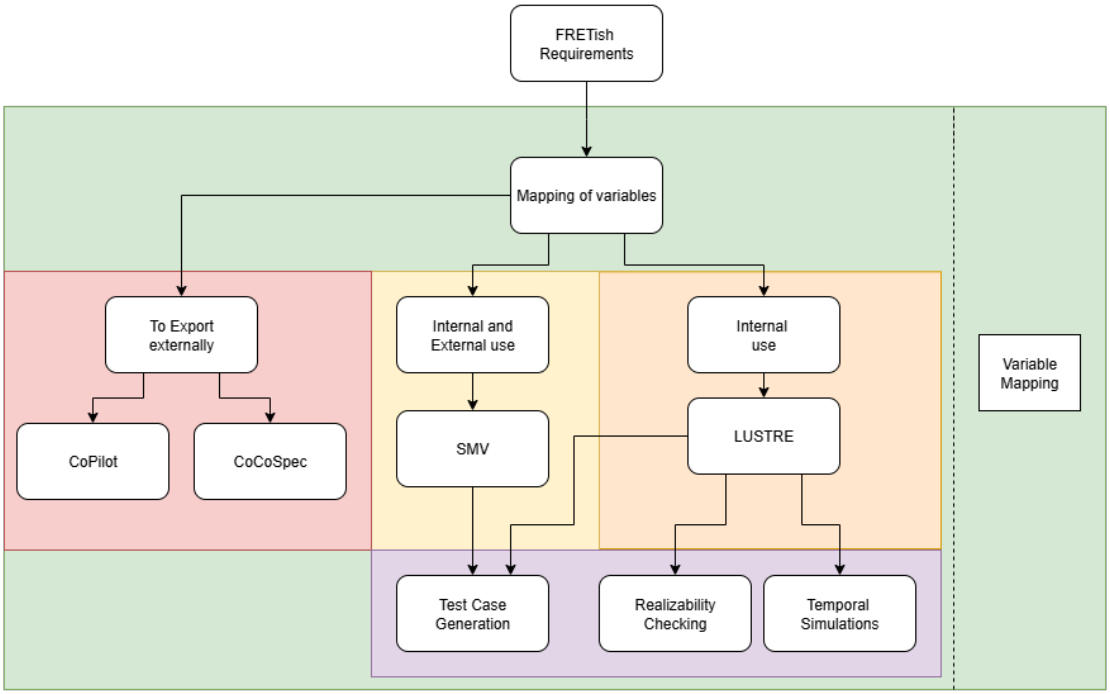


Figure 3.16: Overview of FRET Variable Mapping ramifications

The Variable Mapping menu, is shown in figure 3.17. It presents all project components, and in a drop-down menu of each component (3.18, it has their corresponding conditions, timings and responses. All of them need to be mapped to a formal attribution that can be understood by the mathematical solvers.

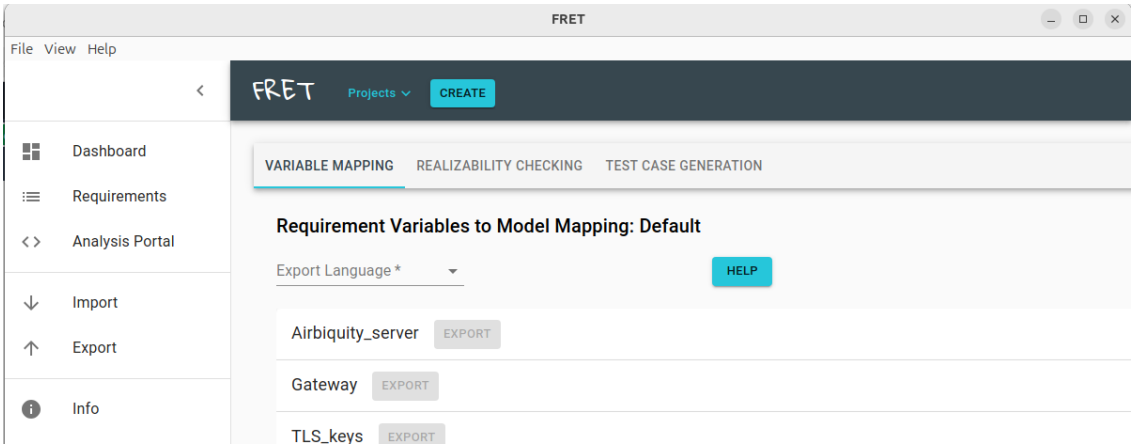


Figure 3.17: FRET variable mapping tab with components

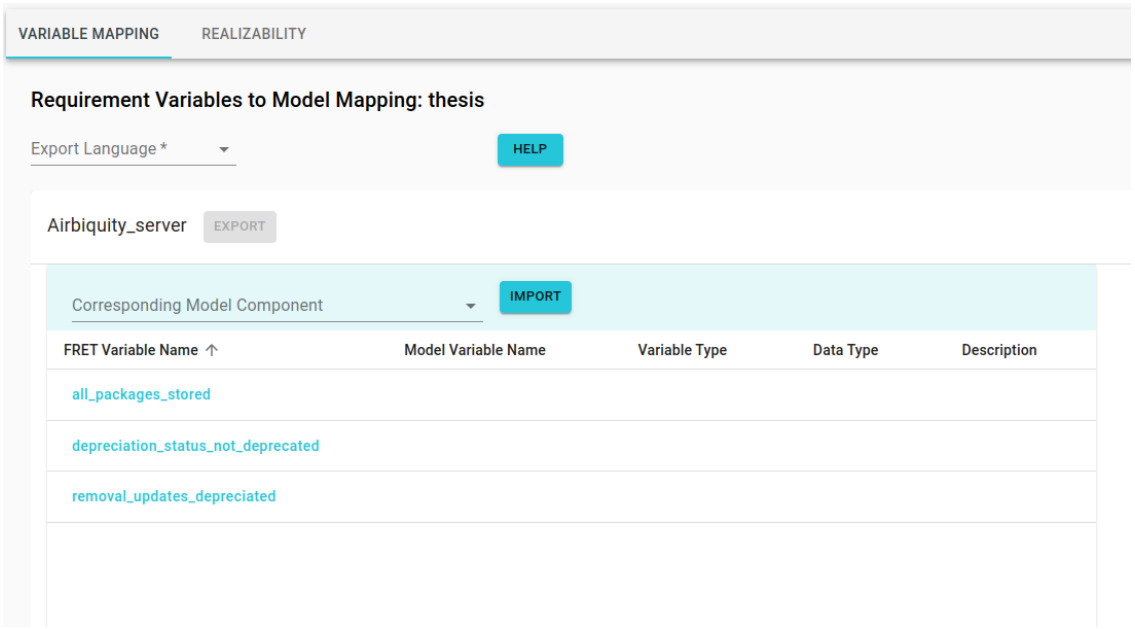


Figure 3.18: FRET components expanded to their corresponding table of variables

From natural language to formal, it is necessary that FRET abstracts variables and properties for checking the realizability and for the simulations. To perform model checking, these abstract elements must be associated with the precise variables and data types, and that is done in this variable mapping menu. For variable types, the options available are table 3.2:

The internal variables are variables that can be computed by the system for further use, meaning they are totally controllable by the system. Their definition can be expanded in Lustre, CoPilot and SMV, by selecting the desired language such as the example of Figure 3.19. Currently SMV attributions are only possible with boolean data types.

Variable Type	Data Type	Description
None	Not Defined	Available for empty exportation to CoPilot
Function	Not Defined	Available for exportation to CoPilot and CoCoSpec
Input	Boolean/ Integer/ Unsigned Integer/ Single/ Double	Represents a system that receives an input from an external source and cannot be modified by FRET
Internal	Boolean/ Integer/ Unsigned Integer/ Single/ Double	Represents any component that can be used, defined, modified and by FRET
Output	Boolean/ Integer/ Unsigned Integer/ Single/ Double	Represents a system that behaves as an output from an external source and cannot be modified by FRET

Table 3.2: Types of variables and data available in Variable Mapping

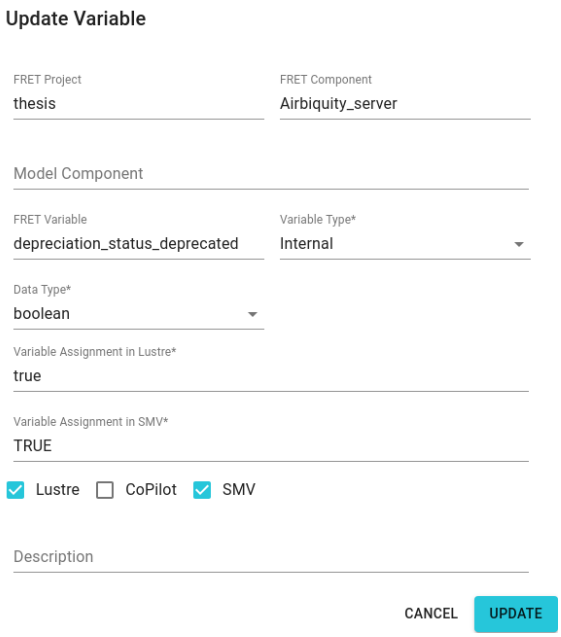


Figure 3.19: Available assignments to internal variables

The Variable Mapping ensures that the requirement is clearly and accurately represented in the formal model and reduces the manual effort and potential errors in translating requirements. After this translation, it brings about seamless integration with formal verification tools, ensuring that the system behaves as intended.

Variable mapping languages exportation

To analyze requirements, FRET allows users to export to other analysis tools such as CoCoSim, Copilot and SMV. Currently, SMV is only possible to use for the test generation exports. In case the user wanting to to the requirements analysis in FRET itself, it needs to have variable mapped to CoCoSpec to use it for the Realizability analysis.

CoCoSpec

CoCoSpec is an extension of the Lustre language that allows specifying how a system behaves in different modes. In FRET is mainly integrated to use it in CoCoSim with Simulink tools. CoCoSpec explicitly models mode-specific behaviors, which helps make the verification process more accurate and scalable. Using CoCoSpec, developers write contracts that describe what each part of the system should do in each mode. These contracts are verified by model checkers like Kind2, which check that the system meets its requirements under all conditions and provide helpful feedback if there are any problems. This approach is particularly useful for verifying complex safety-critical systems that operate in multiple modes, improving reliability and reducing errors early in development [62, 63].

CoPilot

CoPilot language export enables runtime verification of embedded systems through the CoPilot monitoring tool. This tool provides real-time assurance by continuously checking system behavior against specified requirements during operation, detecting deviations from expected behavior. CoPilot is particularly suited for high-reliability and safety-critical domains including aerospace, automotive, and medical devices. At present, CoPilot lacks direct integration with FRET, requiring users to manually configure its runtime environment [64].

SMV

Symbolic Model Verifier (SMV), is implemented in FRET as a direct interface to the NuSMV model checker. It is a workflow for formal verification backend to check whether formalized requirements, originating from FRETish statements, are satisfied by a system model. This integration is essential because SMV model checkers perform exhaustive analysis, exploring all possible behaviors of the system model to search for violations of the desired properties. If a requirement is not satisfied, SMV typically generates a counterexample trace, making it easier for engineers to diagnose problems and refine requirements or models.

3.3.4 Types of analysis

FRET incorporates several built-in and extensible analysis capabilities to ensure the overall quality, clarity, and correctness of specified requirements. These analyses are integral to the requirements engineering process within FRET and are present throughout its usage. The primary types of analysis available in FRET include semantic analysis, realizability analysis, consistency checking, and simulation analysis [1, 60].

Semantic analysis

Semantic analysis is performed when a requirement is created or edited. This analysis checks that each requirement statement is syntactically and logically correct according to the Fretish language templates and patterns. It verifies compliance with the expected formal structure (syntax), semantics, and logical structure ensuring requirements are well-formed and interpretable by the tool. Semantic analysis is built into FRET's core and does not require any external model checker to function. This helps prevent formulation errors from the outset and maintains the quality of the requirements database [1].

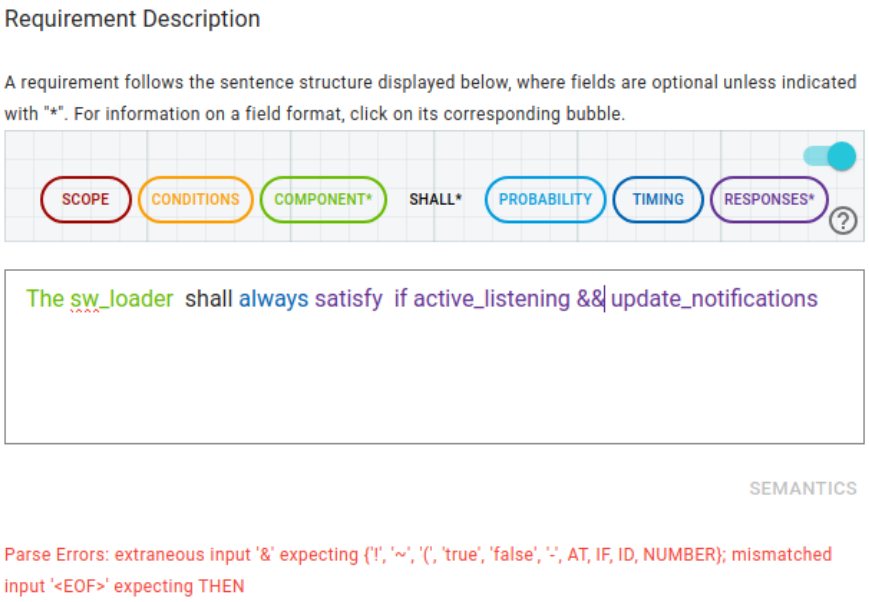


Figure 3.20: FRET: Example of an error on sematic analysis

Realizability Analysis

FRET analyzes the requirements in the project using a process called Realizability Analysis, determining whether the set of requirements can be implemented/achieved or not under all possible conditions. It checks the feasibility, consistency, constraints and the presence of contradicting behaviors. If any of those are caught, FRET flags them. This analysis is fundamental in ensuring that the system does not find itself in a situation where it cannot guarantee the fulfillment of the requirements and can operate in two modes: monolithic or compositional.

In **Monolithic Analysis**, the set of requirements is treated as one complete system, checking whether all requirements can be realized together. Although it provides a great picture of the complete feasibility of the project, it is not scalable for projects that have more than one unique system, and it can take longer computer resources to perform this analysis.

The **Compositional Analysis**, requirements are decomposed into smaller components, based on the individual system outputs. Each component is individually checked for realizability, and if all components are realizable, then the global requirement from where the components broke down is also realizable. If one of the components is not realizable, then it means that the main requirement is also not feasible. When a set of requirements is found infeasible or contradictory, realizability analysis will identify and flag the specific items involved, supporting early error correction [60]. This analysis helps to clarify the dependencies of the requirements and is capable of checking complex and larger systems because it is more scalable.

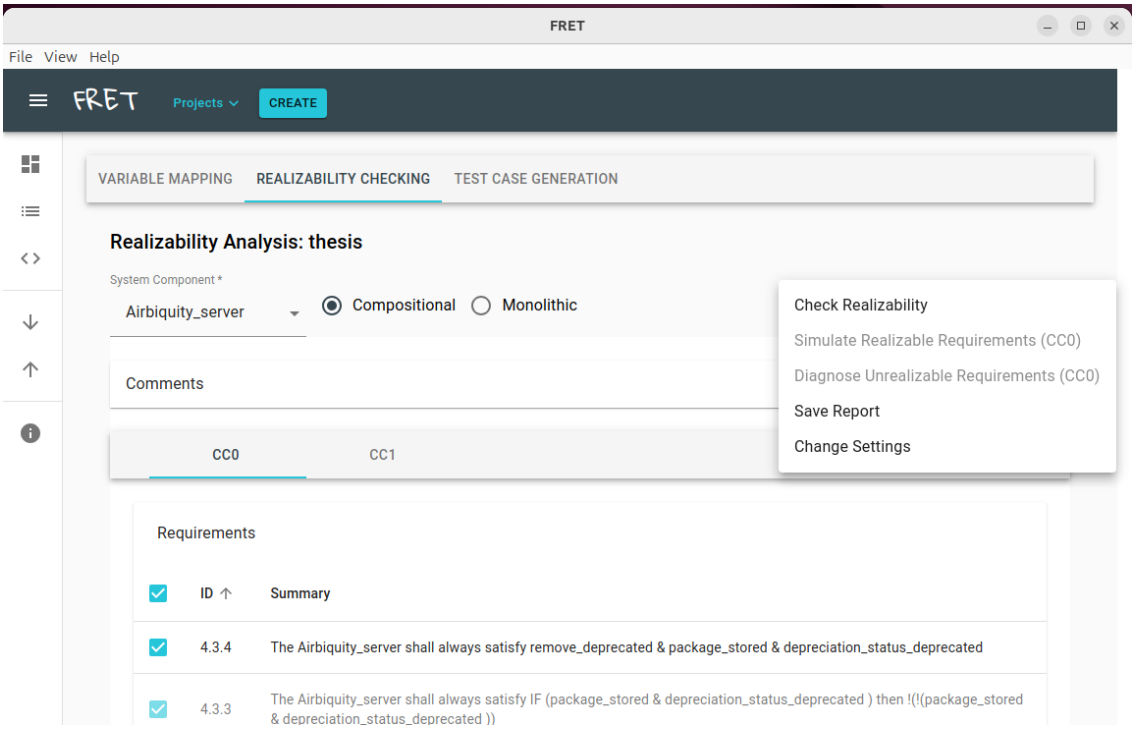


Figure 3.21: Checking realizability of requirements from one component

Consistency checking

Consistency checker is a core aspect of FRET and behaves as a continuous process. Every time a requirement is added, changed, or deleted, this checker verifies the relationships between the requirements to check if there exist any kind of inconsistencies like conflicts or contradictions in them. This includes checking for conflicting constraints, incompatible timing properties, or contradictory system states. . The temporal logic properties of this checker are provided by model checkers like NuSMV or Kind2.

An example of a flagged requirement checker is:

Requirement 1: "If action1 happens, action2 must occur".

Requirement 2: "If action1 happens, action2 must not occur"

These two requirements will be flagged as an inconsistency because for the same occurrence of action1 there exist two opposite behaviors regarding action2. By catching these conflicts early, consistency checking helps avoid late rework and unexpected or unsafe system behaviors. Consistency analysis can use temporal logic properties and leverage connected model checkers for complex logic [1].

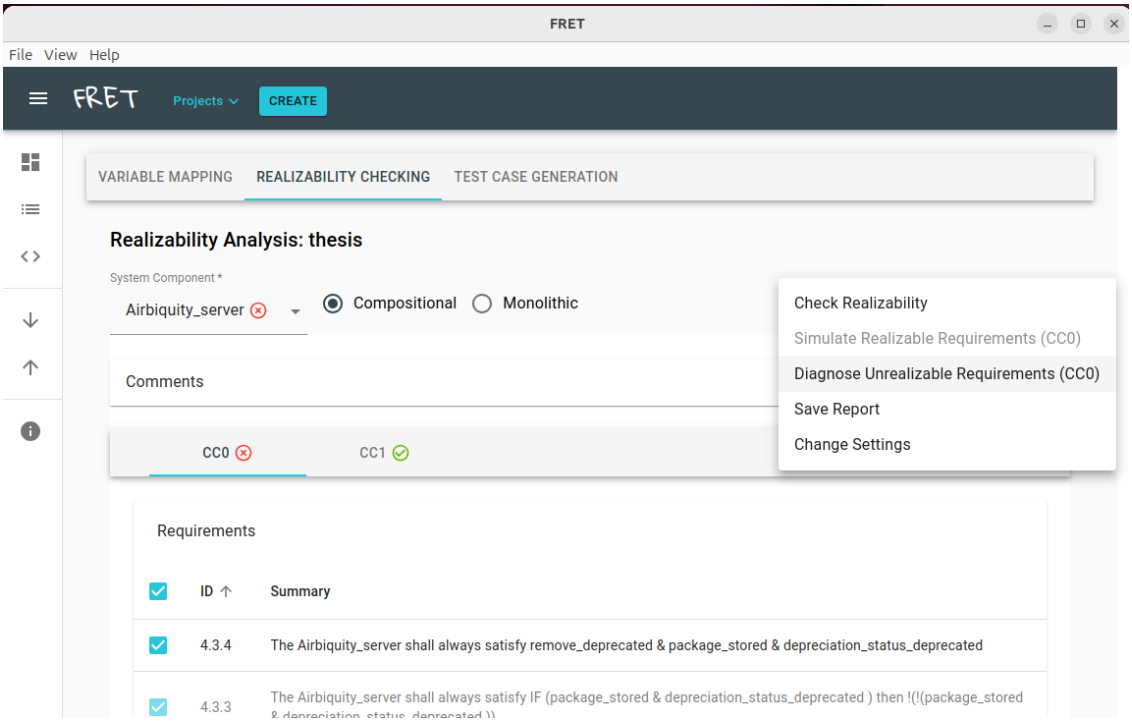


Figure 3.22: Checking consistency between requirements

Simulation Analysis

Simulation analysis allows users to test and visualize requirement behaviors under different scenarios. By creating executable models from the requirements, simulation analysis lets engineers observe system behaviors such as state transitions, timelines, and other sequence visualizations. Simulation is especially important for understanding interactions, bottlenecks and timing dependencies. While basic simulation features are available in FRET, integration with external model checkers such as NuSMV, JKind, Kind2, and LTLsim provides advanced capabilities, particularly for temporal logic simulation and more detailed analysis. Early detection of issues during simulation helps minimize risk and enhance product quality for safety-critical projects [1].

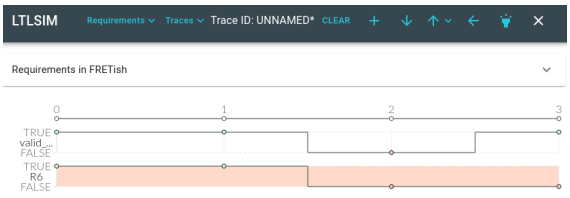


Figure 3.23: Example of a temporal simulation from the requirements

3.3.5 Model Checkers

Model checkers, is where the semantic, realizability and consistency analysis happens. The architecture of FRET is designed to be extensible, supporting the seamless addition of new model checkers as requirements for analysis evolve. These checkers are integrated into FRET to facilitate rigorous analysis of system requirements, ensuring they meet desired standards and can be effectively implemented [1]. Figure 3.24 illustrates how the model checkers JKind

and Kind2 are integrated within FRET, as well as the underlying tools necessary for their operation, including Lustre, Z3, and MBP.

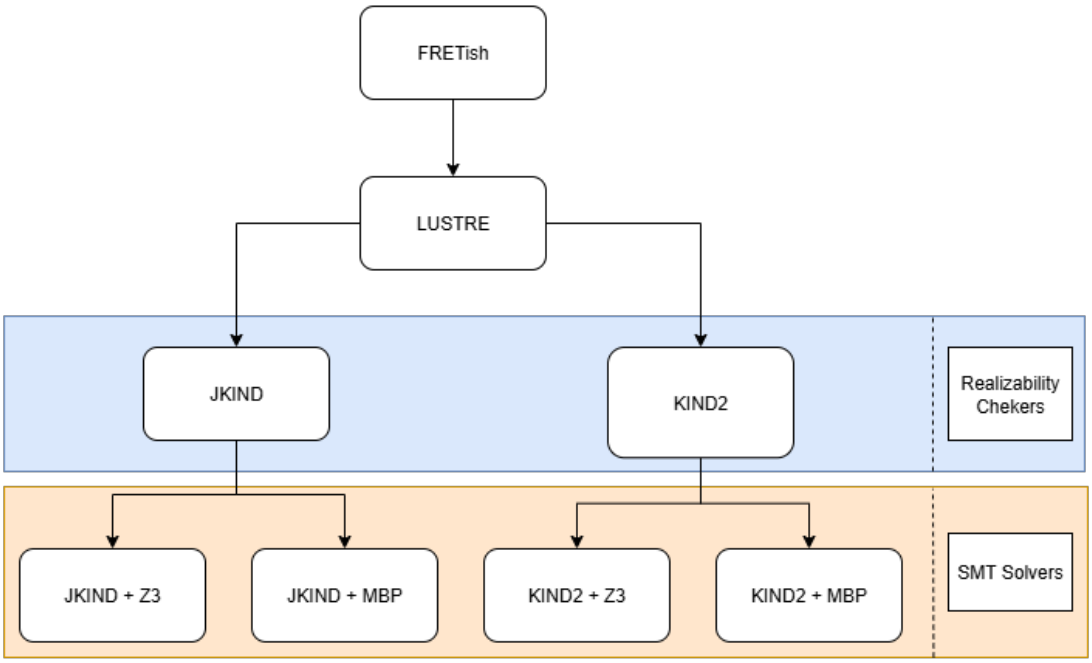


Figure 3.24: FRET model checkers overview

These integrations enable FRET to verify safety and liveness properties, check system assumptions, and ensure temporal logic correctness. Additionally, they generate diagnostic feedback such as counterexamples and verification traces, which FRET surfaces to users by linking them to the relevant requirements. This capability supports iterative refinement and early detection of design errors, making FRET a comprehensive tool for formal requirements engineering.

The following subsections describe how the model checkers currently integrated with FRET.

LUSTRE

Lustre is a synchronous dataflow programming language designed for the specification and analysis of reactive systems. Its declarative nature allows system behavior to be defined through equations relating inputs and outputs over time. This makes Lustre particularly suited for modeling embedded and safety-critical systems where precise timing and deterministic behavior are essential [65].

In FRET, Lustre serves as the intermediate formal language bridging requirements written in the Fretish language and the model checkers. FRET translates natural-language requirements into precise Lustre representations, enabling unambiguous mathematical modeling essential for automated formal verification [65, 66]. This translation is fundamental to making automated verification feasible.

After this translation, FRET workflow can start the formal verification steps using the model checkers JKind or Kind2.

JKind

JKind is a Java-based model checker specializing in the safety analysis of synchronous reactive systems. It operates on formal models usually specified in Lustre. JKind uses Satisfiability Modulo Theories (SMT) solvers to efficiently verify properties [67].

Within FRET, JKind is integrated to perform realizability checking, which determines whether formally specified requirements (translated into Lustre) can be implemented by a system. It verifies properties such as the absence of deadlocks and the avoidance of critical or unsafe states. When verification fails, JKind produces counterexamples and diagnostic information that FRET presents to help users interpret results and refine requirements [65].

JKind appears as an available engine under the Analysis portal's Realizability tab for checking requirement realizability.

Kind2

Kind2 was developed as a successor to JKind, targeting larger and more complex systems through enhanced modular and compositional verification [68]. It functions as an alternative or complementary model checker for Lustre models generated from formalized requirements. After translation into Lustre, Kind2 verifies realizability and correctness of system behaviors specified by requirements contracts.

Kind2 excels at identifying issues both at the system and component levels, making it particularly effective for compositional realizability analysis of distributed or hierarchical architectures. Its contract-based verification methodology breaks down verification tasks, improving scalability for large industrial systems [65, 66]. Like JKind, Kind2 relies heavily on SMT solvers to handle complex logical formulas, and it provides actionable diagnostic information to guide system designers.

Kind2 similarly integrates into FRET's backend and is selectable from the Analysis portal's Realizability tab.

SMT Solvers

For JKind and Kind2 to perform realizability analysis, a solver is needed to discharge the logical constraints arising from formal verification specifications. The integrated solvers used within FRET's toolchain are:

Z3: Z3 is a widely used SMT solver developed by Microsoft Research. It proves or disproves properties of systems by checking satisfiability of logical formulas, applying exact Quantifier Elimination (QE) tactics, which is subject to timeouts. Z3 is the default solver integrated with the JKind and Kind2 engines in FRET. In JKind, QE tactics are often run in parallel to improve efficiency [69].

MBP: Model Based Projection (MBP) functions as a fallback procedure for quantifier elimination within JKind and Kind2. MBP provides an over-approximation of QE needed for realizability checking algorithms. Users can select this solver when this approximation is preferable, balancing performance and precision [69].

Per default, FRET uses the Z3 solver when the user selects "JKind" or "Kind2" in the Realizability tab of the Analysis portal. If the user wants to use the MBP solver it needs to select the "Jkind + MBP" or the "Kind2 + MBP" option.

3.3.6 Simulation Checkers

To perform simulation analysis, FRET integrates specialized simulation checkers that validate elicited requirements by modeling possible behavioral scenarios. These simulations enable early detection of inconsistencies and unfeasible requirements, thereby supporting iterative refining and verification of system specifications [1]. Figure 3.25 illustrates how the temporal simulators LTLsim and NuSMV are integrated within FRET, as well as the underlying processes necessary for their operation.

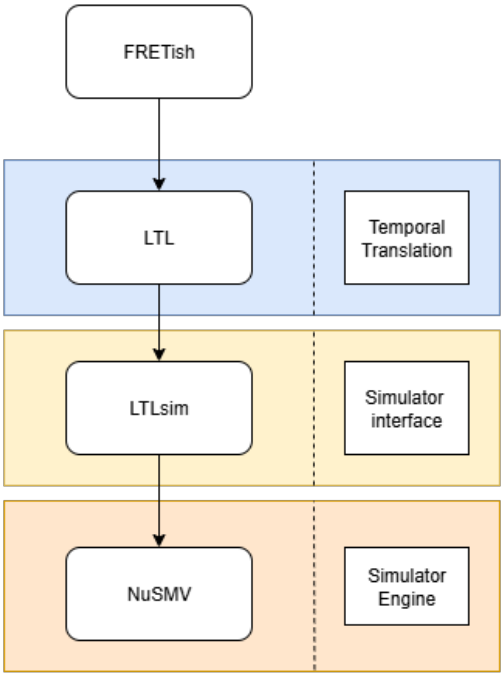


Figure 3.25: FRET model checkers overview

All requirements written in FRET are initially translated from Fretish templates into the Lustre formal language. While model checkers can directly consume Lustre models, simulation checkers require translating the Lustre representations into temporal logic properties, which are then used by simulators to visualize and validate dynamic system behavior.

Temporal properties

The Lustre model incorporates the semantics of Fretish requirements and provides the foundation for translating them into temporal logic formulas that express properties over time. FRET supports Linear Temporal Logic (LTL).

LTL: is a temporal logic used to specify system properties along linear sequences of states, focusing on single execution paths. It includes temporal operators such as "eventually," "always," and "until," which express how properties evolve over time. LTL is particularly effective for specifying constraints on event sequences within reactive systems. In FRET, LTL formalizes temporal requirements, facilitating precise verification and simulation of system behaviors along execution traces [65, 70].

NuSMV

New Symbolic Model Verifier (NuSMV) adopted symbolic model checker for finite-state systems. It supports temporal logic formalisms such as LTL, which are used in FRET's analysis workflow [71, 72].

Requirements specified in the Fretish language are translated into temporal logic formulas such as LTL. NuSMV checks these properties against system models to verify requirement satisfaction and to identify violations of temporal constraints. This verification process helps detect scenarios where the system behavior contradicts specified requirements, particularly those involving timing and ordering constraints crucial for safety-critical systems.

In FRET, NuSMV operates as a backend verification engine essential for time-related analyses. While NuSMV's execution is integrated into FRET's toolchain, it is not exposed directly as an option within the FRET Analysis Portal user interface. This abstraction simplifies user interaction while leveraging NuSMV's powerful verification capabilities behind the scenes [1].

LTLsim

LTLsim is a simulator for LTL formulas integrated into FRET to help users visualize and validate temporal properties interactively. It enables assessment of system behavior over time under various conditions, helping stakeholders understand temporal relationships and verify alignment between specified requirements and actual system dynamics.

The simulator presents simulation results graphically, with simulation time on the horizontal axis and requirement states or conditions on the vertical axis. This layout allows detection of ambiguities or flaws in requirements through scenario-based exploration.

Users access LTLsim via the FRET Analysis Portal by opening the Realizability Analysis tab and selecting "Simulate Realizable Requirements," which opens a window displaying temporal simulation results. LTLsim relies on simulation output files generated by NuSMV to present detailed timing analyses and graphical visualizations [1].

3.3.7 Generation of Test Cases

In FRET version 3.0, the "Test Case Generation" tab was added. This new feature enables users to automatically derive test cases directly from formally specified requirements, streamlining the validation process and strengthening the link between formal requirement analysis and practical system verification. By integrating automated test case generation, FRET ensures test strategies to adequately cover critical requirements, promoting early detection of specification flaws and improving overall system confidence. This advancement further closes the gap between requirements engineering and verification, facilitating more robust and reliable development workflows. Currently, is the test case generation is possible using NuSMV or Kind2.

NuSMV: Test case generation is currently exclusive to requirements that only have boolean type variables defined. Despite this current limitation, it has the advantage of allowing to simulate the generated test cases.

Kind2: Main advantage is that independent of the data type that the component variable has, it always able to generate a test case. However, it does not support any type of generated test case simulation.

Each engine is worth considering, depending on what is goal with the automatic test case generation and the variable mapping done in the project.

It is important to understand how management of complex requirements can be achieved by different tools. The comparison above highlights FRET's unique capability to bridge natural language and formal specification, offering consistency checking, traceability and verification features that are not available in the other tools. In contrast, DOORSTOP, while accessible lacks the comprehensive traceability and the automated validation offered by FRET and DOORS. FRET emphasis on usability, formal rigor, and traceability, presents a well-balanced methodology ideally suited as an upcoming alternative for requirement analysis in safety-critical systems. However, FRET is still a tool under development that still lacks documentation that made the adoption of FRET a bit difficult. The 3.0.0 version major update, despite useful, also required some re-work on the tool analysis and utilization. This release brought major advances including the support for probabilistic requirement specification, the insertion of test case generation capabilities, and several functionality updates to the analysis portal and to the LTLSIM simulator. All these changes represent a significant increase in the tool's capability for expressive specification, automated analysis, and test generation that contribute to a more thorough and scalable formal requirements engineering process.

Chapter 4

Use Case Development

The concept-project that lead to this case study aims to perform a Software Over The Air Update to a classic ECU that only has communications via CAN. While this project is still in a proof of concept phase, the requirements analysis is already mature enough to be used as a study reference for the development of this thesis.

To facilitate a clearer understanding of the requirements under analysis, this chapter elucidates about the concept-project structure. This overview is essential for enhanced comprehension and is only here for information purposes since the goal is not explaining the software development behind.

4.1 Project structure

A modern vehicle has between 70 to 120 ECUs, and not all of them communicate in the same way. Depending on the ASIL classification and/or the typical use case of the ECU some of them only have CAN communication protocol. However, this doesn't mean that the ECU that is only capable to communicate via CAN won't ever need to be updated, and facilitate this update is one of the current challenges in the industry.

The goal was to create a way of updating the software of a classic ECU (this is an ECU that is only capable of UDS/CAN communications) with an OTA update. Since modern vehicles have a mix of different ECU's with different communication protocols, the goal was to use one of the ECU that has the capabilities of receiving an OTA update as a middle-platform to then update the classic ECU. The figure 4.1 represents the visualization of interactions.

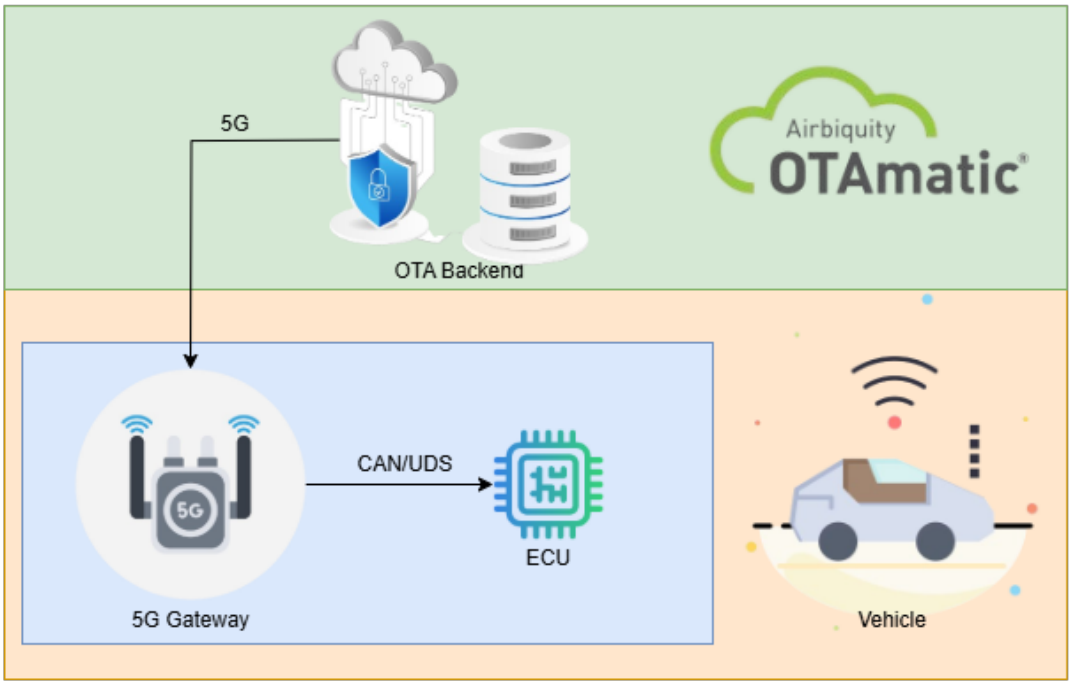


Figure 4.1: Project structure and technologies

When new software becomes available for an update and is ready for deployment, it is sent to Airbiquity, a third-party company collaborating on this concept by providing their OTAmatic Software Management Platform. This platform securely orchestrates and automates the software update campaigns. OTAmatic relies on the UPTANE framework to verify the authenticity and integrity of the incoming software updates.

The system incorporates a specialized ECU called the Gateway, which possesses multiple capabilities including receiving encrypted 5G communications via the TLS protocol. Upon receiving software updates OTA, the Gateway processes and verifies the trustworthiness of the software package before allowing any communication with other ECUs.

To communicate with other ECUs, the Gateway includes a translator module that converts the accepted OTA software from the Diagnostic over Internet Protocol (DoIP) format into Unified Diagnostic Services (UDS). The UDS communication is then transmitted via the Controller Area Network (CAN) bus to a classic ECU, which ultimately performs the software update on that ECU.

4.2 Methodology

This thesis was developed by using a waterfall methodology where the identified steps are:

1. Definition of the details of the use case, the definition of validation scenarios, and definition of a set of requirements, functional and non-functional;
2. List and review approaches to specify requirements using structured natural languages, semi-formal, and formal requirement specification languages. Evaluation of the identified

languages with respect to the defined set of requirements, including identification of gaps;

3. Development of an approach that addressed the identified gaps, and that connects to formal verification tools;
4. Validation of the proposed approach with respect to the use case and the defined scenarios;

The concept-project provides a real-world example to study the different methods of requirement writing. Requirements can be written in natural or formal language, and it is necessary to understand the differences, strengths, and weaknesses of them in an industrial environment. The workflow of the requirements study and analysis is represented in 4.2.

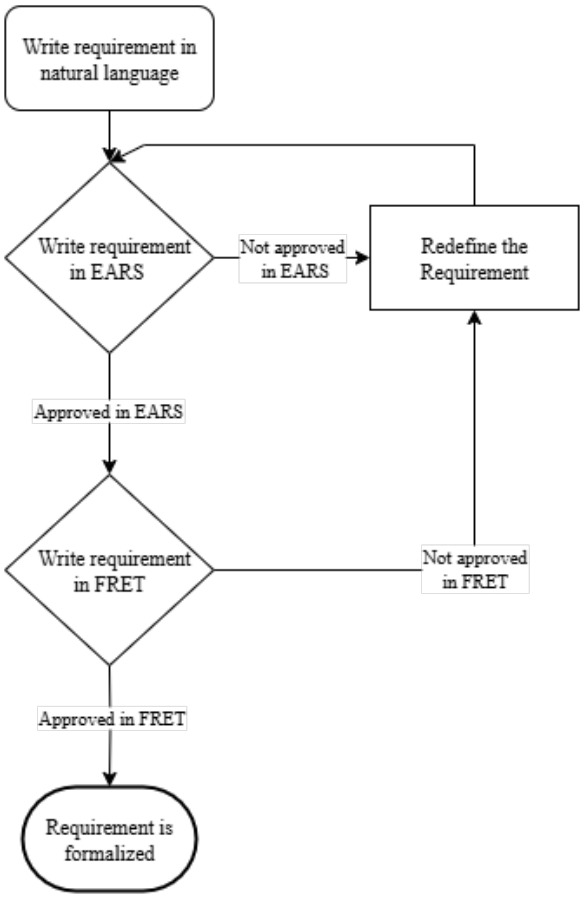


Figure 4.2: Requirement process from natural language to formal

After having the elicitation of the project requirements written in natural language it will be possible to extrapolate to create the ones in formal language. With this approach, a pool of data is created that can be compared, analyzed, and understand the strengths of each approach.

4.3 Requirements analysis

In the realm of software engineering, one of the foundational stages in the development life-cycle is requirements analysis. At its essence, this stage aims to comprehend and articulate what a system should accomplish, serving as the blueprint for subsequent design and implementation phases. However, the challenge often lies in the interpretation and translation of stakeholders often varied and sometimes ambiguous needs into precise specifications.

The initial step in this process involves gathering requirements expressed in everyday language. These requirements are often rich in context but may lack the structure necessary for systematic analysis and validation. It would not be feasible to analyze all the requirements written for the concept-project, so it was only selected the most important ones from each component, keeping in mind that they could all be interconnected and maintain a general idea regarding the project goal.

Then from the selected list of requirements, those will be re-written using the structured approach of EARS (Easy Approach to Requirements Syntax) patterns. Since EARS provides a systematic method for transforming natural language requirements into a more structured representation. By decomposing requirements into essential elements such as actions, actors, and objects, EARS patterns facilitate clearer communication and enhance the comprehensibility and manageability of requirements.

Once requirements are formalized into EARS patterns, the focus shifts to the analysis phase. Here, the aim is to ensure that the requirements are complete, consistent, and unambiguous. To aid in this endeavor, it will be used FRET (Formal Requirements Elicitation Tool). FRET offers capabilities for analyzing requirements, identifying dependencies, detecting conflicts, and verifying correctness, thus mitigating the risk of misunderstandings or oversights.

This thesis pretends to analyze different methods on requirement analysis and check what each method is more beneficial and how to apply it. From the use case that is being explored the work is divided in requirements selection, requirement analysis, and methods and tools comparison.

4.4 Requirements selection

From the case study, it was selected a few of the most interesting requirements to analyze. The selection was made considering requirements from several classes: architectural, temporal, and functional. These requirements will be explored in the whole document and be mindful that those requirement at more at stakeholder lever than in deep technical one.

Requirements list:

1. To validate the package received, it shall do a verification via TLS and only after Airbiquity will send it to the software loader application.
2. The software loader shall offer the possibility to update any new software update available for the vehicle.
3. The system should offer the functionality of SW update confirmation before the OTA update is flashed onto its target.
4. If the available package update is deprecated, the update is no longer available.

5. The download of the package is done only on reliable networks and if the package size is 30% less than the total ROM of the ECU.
6. The authenticity of the Airbiquity OTA package update shall be guarantied using TLS encryption communication.
7. The OTA Orchestrator shall be activated and tested by using Airbiquity.
8. The 5G gateway receives the update and it verifies if the package is trustworthy, then translates it and updates the ECU.
9. The update shall be done using the in-place method, enhancing the need to be aware of how much memory it has available.

Requirements breakdown:

Requirement	Breakdown
1	Package validation verifies the TLS key. After TLS key is approved sends the package to software loader
2	Software Loader receives information of a new update. Software Loader shall present to the user the update possibility.
3	No update shall be done without a user confirmation. After user confirmation the OTA update is valid to proceed.
4	Airbiquity shall know the package deprecated status. When package is deprecated it should be removed from the available update list.
5	If network available is not reliable the download will not start. When the network is reliable, it checks if the package size. If the package size is less than 30% of the total ROM of the ECU, then it allows to download.
6	A SW package is only valid when the TLS key is provided by a valid entity.
7	If Airbiquity server is disconnected then the OTA Orchestrator is not active.
8	The update is received by 5G to the gateway from the Airbiquity server. If the package is trustworthy it will be translated from DoIP to UDS communication by the gateway. After that translation is well succeeded it will be flashed on the ECU via CAN.
9	Before the update starts, the gateway confirms the current ROM memory space available in the ECU. If the ROM memory has enough space for the update, it will do it using the in-place method. If the ROM memory has does not have enough space for the update, it won't proceed with the update.

Table 4.1: Breakdown of the requirements list into smaller requirements

Now that we have a first requirements analysis breakdown, they are still too high level for our application. From this breakdown there were derived 20 requirements but not all of them are

totally clear and they have omissions. The next step is to introduce a semiformal pattern to allow the requirements analysis to proceed with more rigor and reduce the possible oversights.

4.5 Requirements in EARS

The EARS patterns were introduced in chapter 2. As a middle point on transforming the requirements written in natural language to FRETish, it is presented their EARS transformation.

Req. ID	Classification	EARS pattern
1.1	Ubiquitous	The <gateway> shall < validate if TLS key is approved>
1.2	Event-Driven	WHEN <package available> and <package is approved> the <gateway> shall <send the software package to the software loader>
2.1	Ubiquitous	The <software loader> shall <be listening to notifications of new updates>
2.2	Event-Driven	WHEN <new update notification is received> the<software loader> shall <present update to the user>
3.1	Ubiquitous	The <software updater> shall <never update without user notification>
3.2	Event-Driven	WHEN <update available> and the<user confirms> the <software loader> shall <proceed with update>
4.1	Optional Feature	WHERE <all packages stored> the <Airbiquity server> shall <check depreciation status>
4.2	Ubiquitous	The <Airbiquity server> shall <remove all deprecated updates>
5.1	Unwanted Behavior	IF <reliable network> <is not available>, THEN the <gateway> shall <prohibit the download of package>
5.2	State-Driven	WHILE <reliable network> the <gateway> shall <check package size>
5.3	Unwanted Behavior	IF <package size checked> and is <less than 70% ROM >, THEN the <gateway> shall <download package>
6	Ubiquitous	The <TLS keys> shall be <provided by a valid entity>
7	Unwanted Behavior	IF <Airbiquity server> <is disconnected>, THEN the <gateway> shall <deactivate>
8.1	Ubiquitous	The <gateway> shall <receive the update package from the Airbiquityserver>
8.2	State-Driven	WHILE <package trustworthy> the <gateway> shall <translate from DoIP to UDS communication>
8.3	Event-Driven	WHEN <translated successfully> and <starts CAN transmission> the <gateway> shall <flash the software on ECU>
9.1	Event-Driven	WHEN <user confirms the update> the <gateway> shall <check ECU ROM space>
9.2	Event-Driven	WHEN <ROM space> <has enough memory> the <gateway> shall <start the update>

Continues on next page...

Req. ID	Classification	EARS pattern
9.3	Unwanted Behavior	IF <ROM space> <does not have enough memory>, THEN the <gateway> shall <cancel update>
9.4	Event-Driven	WHEN <update accepted> <and update has started> the <gateway> shall <use the in-place method>

Table 4.2: Requirements in EARS

This table 4.2 shows all final requirements validated in EARS. This validation is concluded with the fact that these patterns are converted into FRETish templates presented in the next section.

4.6 Requirements in FRET

The FRET templates (FRETish) were introduced in chapter 3. The requirements written to FRETish are presented in table 4.3.

Req. ID	Classification	FRETish
1.1	Prescribe Format	The Gateway shall always satisfy TLS_key_approved
1.2	State Transition	Upon (package_available & package_approved) the Gateway shall at the next timepoint satisfy send_sw_package_to_sw_loader
2.1	Prescribe Format	The sw_loader shall always satisfy if active_listening then update_notifications
2.2	Process Command	Upon update_notification_received the sw_loader shall immediately satisfy present_to_user
3.1	Prescribe Format	sw_updater shall always satisfy user_notification
3.2	Change State	sw_loader shall always satisfy if (user_confirmed & update_available) then update_proceed
4.1	Process Command	Upon all_packages_stored the Airbiquity_server shall always satisfy depreciation_status_not_deprecated
4.2	Prescribe Format	Airbiquity_server shall always satisfy removal_updates_depreciated
5.1	Set Diagnostic Flag	Upon unrealiable_network the Gateway shall immediately satisfy !sw_download
5.2	Process Command	Upon reliable_network the Gateway shall always satisfy verify_package_size
5.3	Check Bounds	The sw_package_size shall always satisfy Rom_size_percentage < 70
6	Prescribe Format	TLS_keys shall always satisfy valid_entity
7	Set Diagnostic Flag	Upon Airbiquity_server_disconnected the Gateway shall immediately satisfy deactivated
8.1	Process Command	Upon Airbiquity_server_sends_package the Gateway shall within 60 seconds satisfy recieve_update_package
8.2	State Transition	Upon (package_recieved & package_trustworthy) the Gateway shall at the next timepoint satisfy translation_from_DoIP_to_UDS
Continues on next page...		

Req. ID	Classification	FRETish
8.3	Process Command	Upon translation_successful_from_DoIP_to_UDS the Gateway shall always satisfy start_CAN_transmission & ECU_SW_flashing
9.1	State Transition	Upon (update_available & user_accepts) Gateway shall at the next timepoint satisfy ECU_ROM_available
9.2	Set Diagnostic Flag	If enough_ROM_available the Gateway shall immediately satisfy start_sw_update
9.3	State Transition	Upon (check_ROM_size & !enough_ROM_memory) Gateway shall at the next timepoint satisfy !start_sw_update
9.4	Process Command	Upon update_started the gateway shall always satisfy use_inPlace_method

Table 4.3: Translated EARS requirements into FRETish requirements

4.7 Requirements comparison in EARS vs FRET

To evaluate the effectiveness and formal rigor of requirement specification methods, this study compares the set of 20 requirements formulated using EARS approach with those formalized in FRET. Despite EARS having only 5 patterns and FRET has more (even that were only used 6 FRETish templates), it was mapped in a 1 to 1 ratio the best correspondence between them. In table 4.4 it shows the mapping and the total requirements inserted in each category.

EARS Category	Count	FRET Category	Count	Mapping
Ubiquitous	6	Prescribe Format	6	Irrefutable system properties
Event-Driven	7	Process Command	5	Triggered by events
State-Driven	2	State Transition	4	Trigger of state changes
Unwanted Behavior	4	Check Bounds	1	Restricts forbidden behavior
Optional Feature	1	Set Diagnostic Flag	3	Models optional capabilities
Not Applicable	-	Change State	1	Consequences of change

Table 4.4: Comparison of requirement category distributions between EARS and FRET.

FRET provides an extended set of templates than EARS, allowing for more precise and nuanced expression of requirements. This greater expressiveness better captures the complexity of real projects, enabling more accurate and comprehensive formalization and analysis of system behavior. The table 4.5 is comparing the requirements themselves in both specification patterns.

Req. ID	EARS pattern	FRETish
1.1	The <gateway> shall < validate if TLS key is approved>	The Gateway shall always satisfy TLS_key_approved
Continues on next page...		

Req. ID	EARS pattern	FRETish
1.2	WHEN <package available> and <package is approved> the <gateway> shall <send the software package to the software loader>	Upon (package_available & package_approved) the Gateway shall at the next timepoint satisfy send_sw_package_to_sw_loader
2.1	The <software loader> shall <be listening to notifications of new updates>	The sw_loader shall always satisfy if active_listening then update_notifications
2.2	WHEN <new update notification is received> the<software loader> shall <present update to the user>	Upon update_notification_received the sw_loader shall immediately satisfy present_to_user
3.1	The <software updater> shall <never update without user notification>	sw_updater shall always satisfy user_notification
3.2	WHEN <update available> and the<user confirms> the <software loader> shall <proceed with update>	sw_loader shall always satisfy if (user_confirmed & update_available) then update_proceed
4.1	WHERE <all packages stored> the <Airbiquity server> shall <check depreciation status>	Upon all_packages_stored the Airbiquity_server shall always satisfy depreciation_status_not_deprecated
4.2	The <Airbiquity server> shall <remove all deprecated updates>	Airbiquity_server shall always satisfy removal_updates_depreciated
5.1	IF <reliable network> <is not available>, THEN the <gateway> shall <prohibit the download of package>	Upon unrealiable_network the Gateway shall immediately satisfy !sw_download
5.2	WHILE <reliable network> the <gateway> shall <check package size>	Upon reliable_network the Gateway shall always satisfy verify_package_size
5.3	IF <package size checked> and is <less than 70% ROM >, THEN the <gateway> shall <download package>	The sw_package_size shall always satisfy Rom_size_percentage < 70
6	The <TLS keys> shall be <provided by a valid entity>	TLS_keys shall always satisfy valid_entity
7	IF <Airbiquity server> <is disconnected>, THEN the <gateway> shall <deactivate>	Upon Airbiquity_server_disconnected the Gateway shall immediately satisfy deactivated
8.1	The <gateway> shall <receive the update package from the Airbiquity-server>	Upon Airbiquity_server_sends_package the Gateway shall within 60 seconds satisfy recieve_update_package
8.2	WHILE <package trustworthy> the <gateway> shall <translate from DoIP to UDS communication>	Upon (package_recieved & package_trustworthy) the Gateway shall at the next timepoint satisfy translation_from_DoIP_to_UDS
8.3	WHEN <translated successfully> and <starts CAN transmission> the <gateway> shall <flash the software on ECU>	Upon translation_successful_from_DoIP_to_UDS the Gateway shall always satisfy start_CAN_transmission & ECU_SW_flashing

Continues on next page...

Req. ID	EARS pattern	FRETish
9.1	WHEN <user confirms the update> the <gateway> shall <check ECU ROM space>	Upon (update_available & user_accepts) Gateway shall at the next timepoint satisfy ECU_ROM_available
9.2	WHEN <ROM space> <has enough memory> the <gateway> shall <start the update>	If enough_ROM_available the Gateway shall immediately satisfy start_sw_update
9.3	IF <ROM space> <does not have enough memory>, THEN the <gateway> shall <cancel update>	Upon (check_ROM_size & !enough_ROM_memory) Gateway shall at the next timepoint satisfy !start_sw_update
9.4	WHEN <update accepted> <and update has started> the <gateway> shall <use the in-place method>	Upon update_started the gateway shall always satisfy use_inPlace_method

Table 4.5: Translated EARS requirements into FRETish requirements

4.8 Requirements overlooked aspects revealed by FRET

The comparison between the original natural language requirements and their formalized counterparts in FRET reveals critical omissions common in informal specifications. The table 4.6 summarizes, for each requirement, the key elements that were either implicit or missing in the natural language version: such as system ownership, specific triggers, preconditions, and explicit system responses. This highlights how FRET's structured approach mandates their clear specification. By requiring the user to define these details explicitly, FRET improves the precision, unambiguity, and completeness of requirements. This formalization not only aids a thorough system understanding but also creates a solid foundation for subsequent verification and validation activities.

Req. ID	Omission in Natural Language	Specified in FRET
1.1	System responsible for TLS validation not specified	Gateway explicitly identified as responsible for TLS key validation
1.2	Preconditions and trigger not identified	Trigger: package approval, system: Gateway, next action: send to software loader
2.1	System reaction vague	System (sw_loader) reaction: receive update notifications specified clearly
2.2	Timing and reaction to update notification missing	Immediate user presentation of notification upon reception defined
3.1	User role implicit	User explicitly defined as actor accepting update notification
3.2	Preconditions and triggers implicit	Update availability and user confirmation explicitly required before starting update
4.1	Ownership and process unclear	Airbiquity server defined as checking depreciation of stored packages
4.2	Action on deprecated update unspecific	Clear removal of deprecated updates by Airbiquity server specified

Continued on next page

Req. ID	Omission in Natural Language	Specified in FRET
5.1	Network conditions implicit	Unreliable network explicitly triggers prohibition of download
5.2	Size check process unspecified	On reliable network, system explicitly verifies package size
5.3	Package size margin unstated	Bound check specified: package size less than 70% ROM
6.0	TLS entity validity implied	TLS keys explicitly validated as authorized entities
7.0	Effect of server disconnection vague	OTA Orchestrator deactivated upon Airbiquity server disconnection
8.1	Reception timing unstated	Gateway receiving update package with 60s deadline specified
8.2	Translation dependency implicit	Translation from DoIP to UDS defined immediately after trustworthy package received
8.3	Communication and flashing sequence implicit	CAN transmission start and ECU software flashing explicitly commanded after translation
9.1	ROM availability check trigger missing	Triggered by update availability and user acceptance, ROM check is explicit
9.2	ROM sufficiency effect implicit	Enough ROM triggers immediate update start
9.3	Negative ROM condition unexplained	Insufficient memory explicitly prevents update
9.4	Update method unspecified	Use of in-place update method mandated once update begins

Table 4.6: Omissions in natural language requirements and their explicit specification in FRET

The most frequent omission in the natural language requirements was the explicitly specification of the system or component responsible for a requirement. For example, natural language statements might describe actions such as verification or update delivery without clarifying which subsystem executes them. Missing this information can lead to ambiguities and inconsistent interpretations.

Many requirements lacked explicit preconditions (e.g., network reliability, user confirmation) or clear event triggers initiating system responses. These details are critical in determining when and how system actions occur, affecting timing, ordering, and fault handling.

Another absence on the untreated requirements was the explicit definition of states or status conditions (e.g., package trustworthiness, update depreciation) that reduced the precision with which behaviors can be understood, monitored, or controlled.

By using FRET, users are guided through structured elicitation that prompts explicit specification of previously overlooked elements. This interactive approach requires users to assign system ownership, define triggers/preconditions, and describe state transitions or changes clearly and earlier in the project. The outcome is an unambiguous requirements model that supports rigorous formal analysis.

This structured formalization helps bridge the gap between stakeholder intent and technical specification, reducing misunderstandings and enabling scalable verification. The disciplined methodology imposed by FRET mitigates risks associated with vague or incomplete natural language requirements, making it a valuable tool in the development of safety-critical and complex embedded systems.

With all requirements successfully mapped in FRET, the focus shifts to leveraging the tool's capabilities to deepen the analysis. Beyond simply clarifying and formalizing requirements, FRET is capable of doing consistency checks, realizability analysis, and test generation, thereby advancing the rigor and completeness of the verification process.

4.9 FRET variable mapping

The variable mapping section defines the correspondence between system components and the variables used within FRET. This mapping is essential to ground the formal requirements to concrete elements of the system, enabling effective utilization of FRET's automated verification and analysis capabilities. By accurately linking variables to their real-world counterparts, the tool can perform consistency checks, realizability analysis, and test generation based on a faithful representation of the system architecture.

The project had 6 systems (called Components, shown in figure 4.3) that were necessary to be mapped.

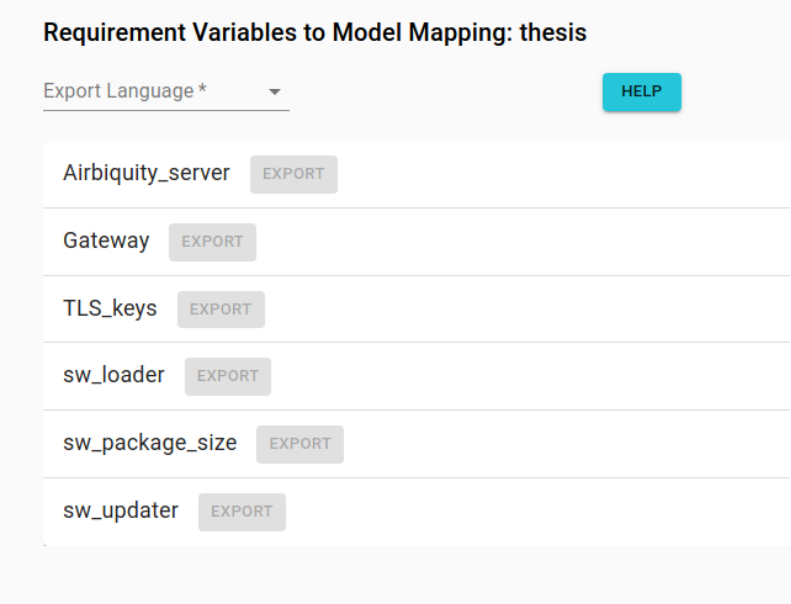


Figure 4.3: All project components

Almost all variables (the project has some exceptions that will be analyzed in detail further), were mapped as being Internal with a boolean data type. This decision was made to be able to use Jkind+Z3 for realizability checking that would allow the further use of the realizability checker simulator and the use of the diagnostic tool for unrealizable requirements. It was also taken in consideration, that by using internal boolean variables, all the features from the test case generation tab would be available to explore. This decision meant that the requirements

were represented in purely boolean logic. As an example, in figure 4.4 is possible to see all variables related to the component sw_loader to be of boolean type.

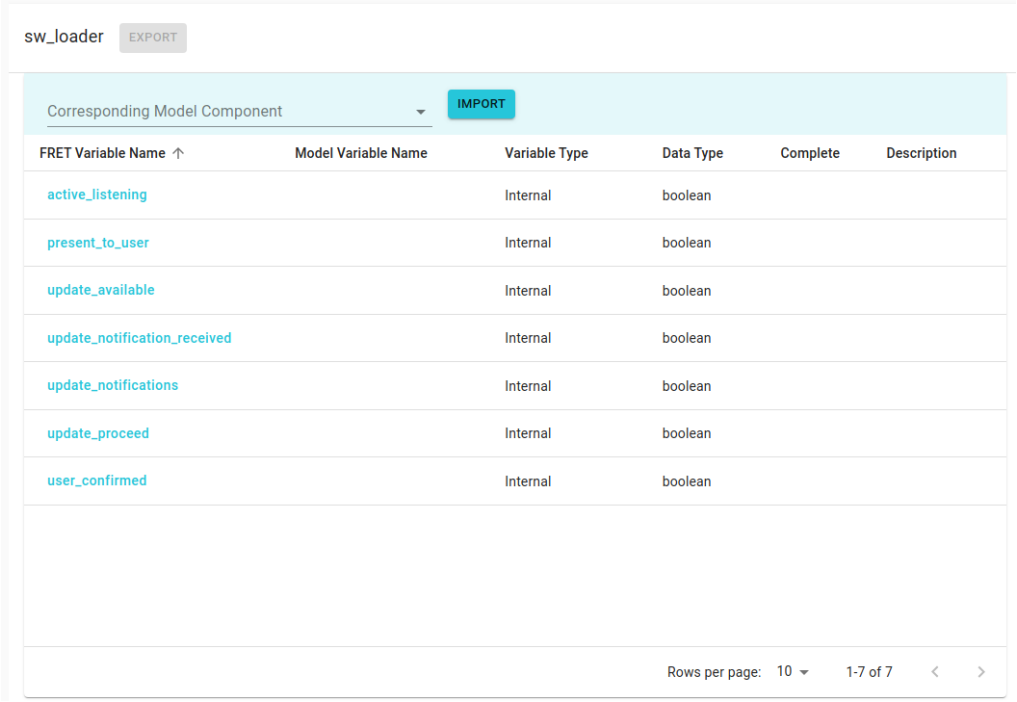


Figure 4.4: Variables mapped as boolean

With all system components and variables mapped, is possible to advance to other verification processes. The next step in this workflow is realizability checking, where the tool assesses whether there exist feasible implementations that can satisfy all specified requirements under the mapped system variables. This analysis helps identify inconsistencies or conflicts early.

4.10 FRET Realizability Checking

Realizability checking is a key formal verification step that evaluates whether the set of specified requirements can be implemented by a concrete system. This process systematically analyzes the formal model to detect contradictory or unachievable conditions, providing early feedback on potential design flaws. By identifying unrealizable requirements, the tool assists stakeholders in refining specifications to ensure a consistent and feasible system design To show an through exploration of the FRET tool, its shown two examples of realizability: a successful one and an unsuccessful.

4.10.1 Example of a Realizable requirement

The requirement in study is the requirement number 6, mentioned in table 4.3. This requirement validates if the TLS_key that encrypts the software package has a valid entity. The first step was testing if the requirement was realizable, and FRET shows in figure 4.5 that it is and that took 0.302 second to evaluate it using the JKind engine.

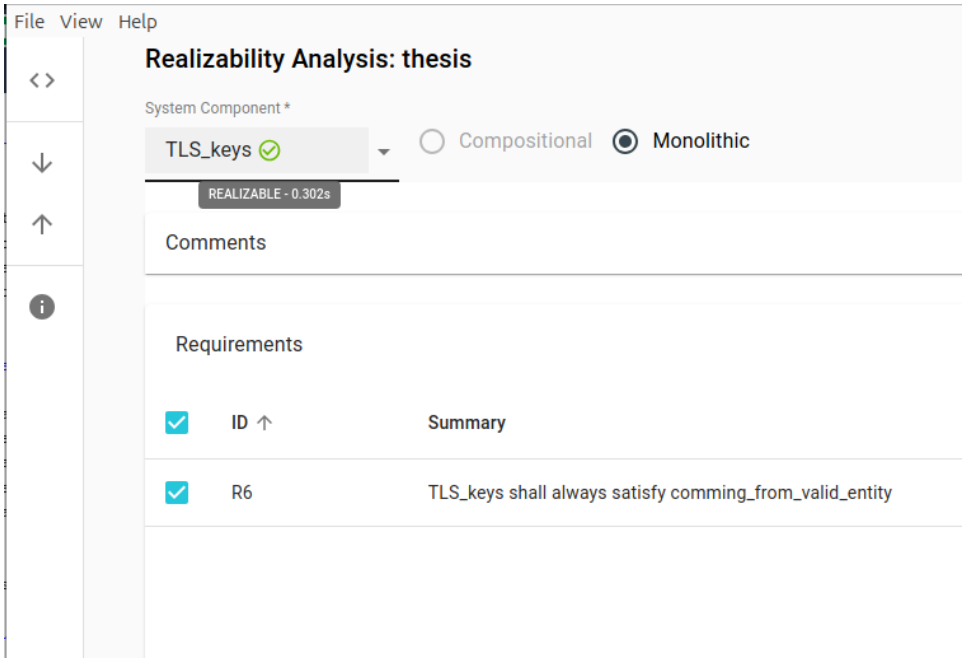


Figure 4.5: Successfully realizability evaluation

Then, it was done the realizability simulation of this requirement, and in figure 4.6 shows, in time, with more than 1 iteration that when the variable "comming_from_valid_entity" will be always valid in time if the value of the variable holds "true".

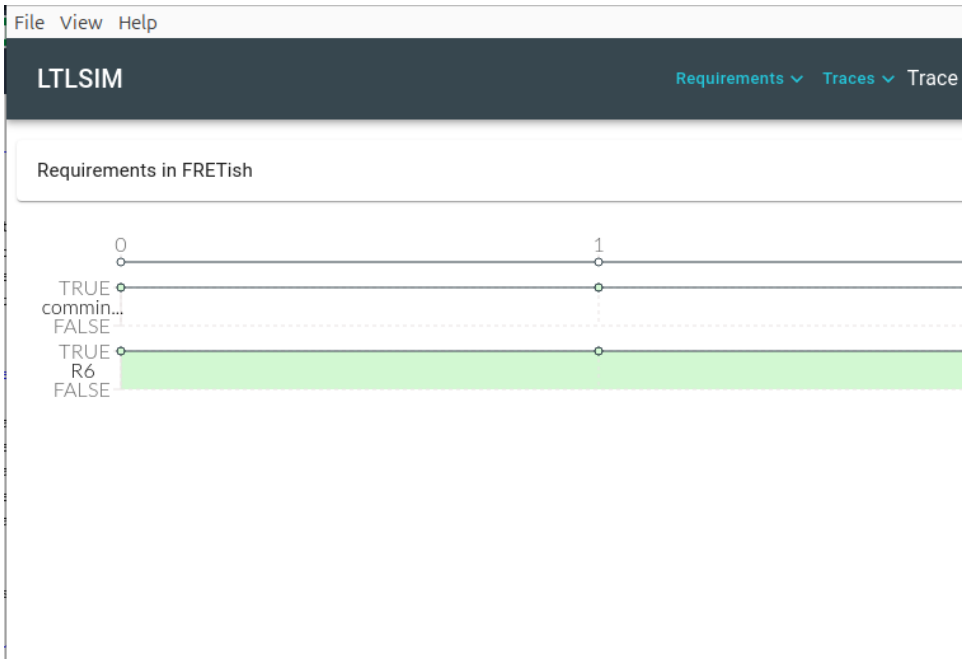


Figure 4.6: Successfully realizability simulation

The LTLsim, also allows the user to directly change the variables values, testing in real time how changes affect the realizability in different points of time, such as shown in figure 4.7.

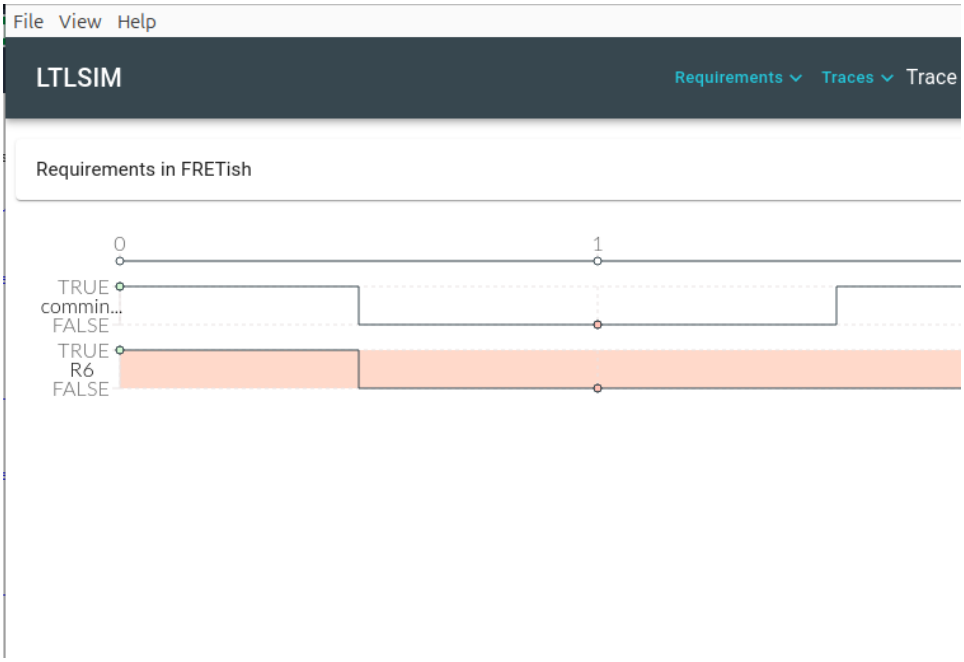


Figure 4.7: Changing outcome of a previously successfully realizability simulation

4.10.2 Example of a Unrealizable requirement

The requirement in study is the requirement number 5.3, mentioned in table 4.3. This requirement validates if the sw_package_size has a size of less than 70% of the ECU_ROM_Size. This requirement is important to evaluate that a received software is not bigger than the ROM of the ECU with a buffer of 30% space available. The first step was testing if the requirement was realizable, and FRET shows in figure 4.8 that the requirement as it is stated is not feasible, it took 0.136 second to evaluate it using the JKind engine.

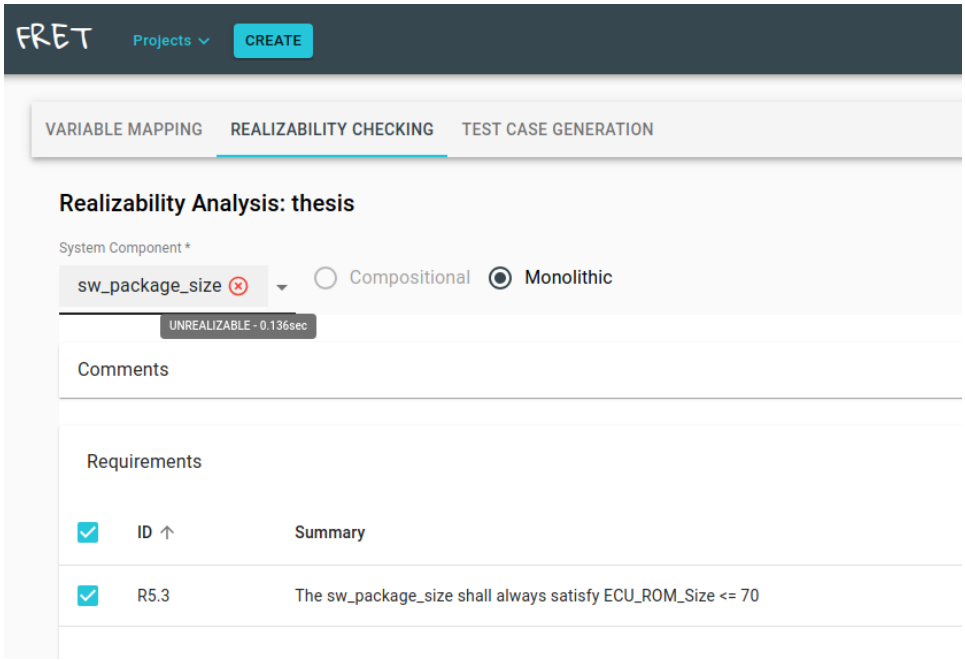


Figure 4.8: Unsuccessfully realizability evaluation

Since the requirement failed the realizability test, is possible to use the diagnostic engine to understand the reason of the failure. The figure 4.9 shows the conflict of the requirement, and all variable types and attributed values.

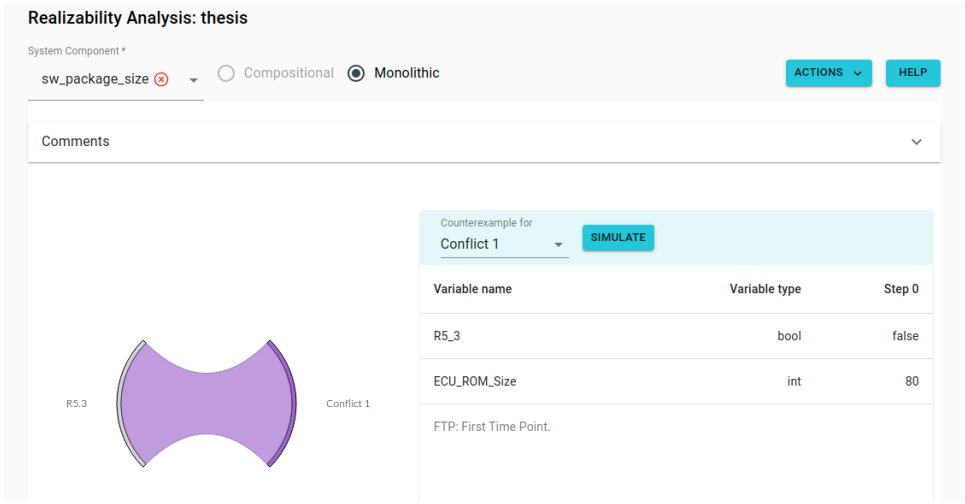


Figure 4.9: Diagnostic result of unrealizable requirement

From the table presented, is already possible to understand that the problem is the ECU_ROM_Size is 80 instead of the 70 mentioned in the requirement. However, the diagnostic tool also allows the user to use the simulation from LTLsim, to understand the problem, and figure 4.10 shows the result.

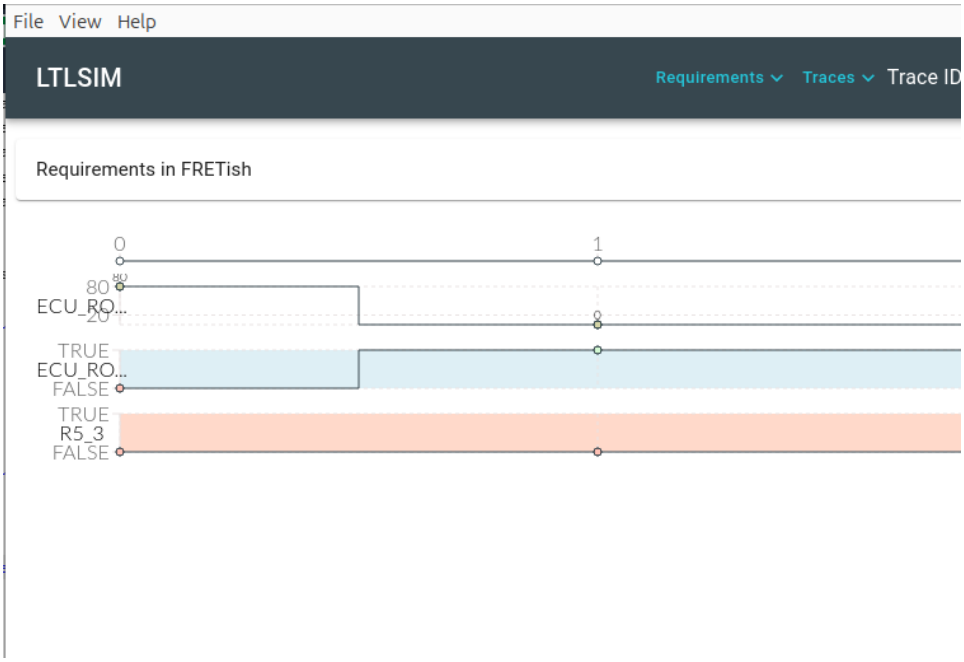


Figure 4.10: simulation result of unrealizable requirement

This failed simulation, shows that already in iteration 1, the requirement is not holding true, so it fails immediately. Since the type of variable in question is an integer, with the value equal to 80, the user can change this value directly to another integer value. Making this change directly in the simulation allows a visual understanding of the threshold of the ECU_ROM_Size variable makes the requirement be realizable or unrealizable. In figure 4.11 it shows directly that a value of 50 in the ECU_ROM_Size variable transforms the unrealizable requirement into a realizable one.

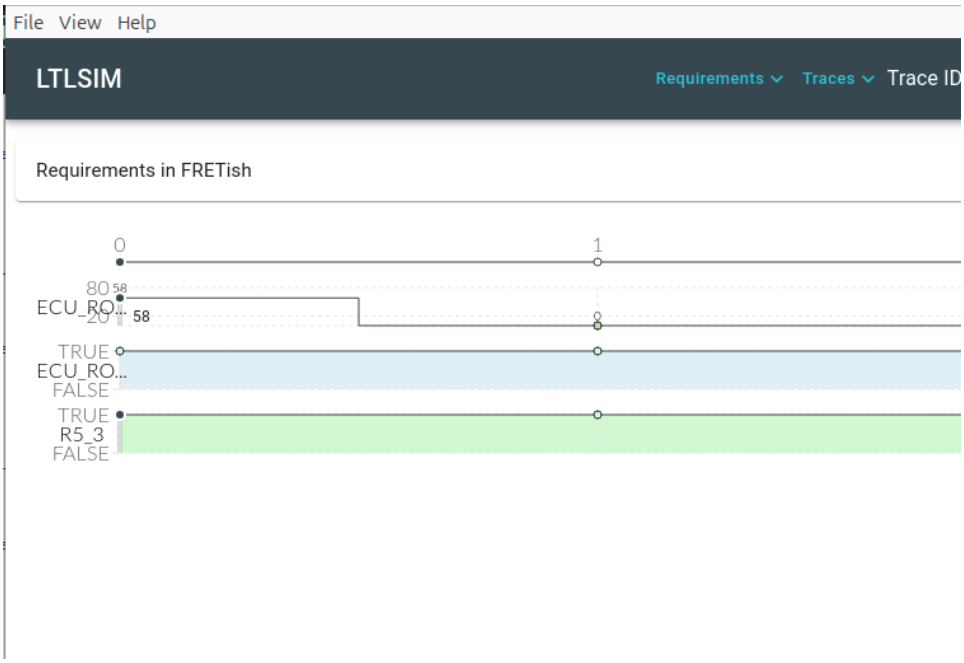


Figure 4.11: Transformation of an unrealizable to realizable simulation

4.11 FRET Test Case Generation

The Test Case Generation tab in FRET represents a significant advancement in connecting formal requirements with practical verification activities. This feature automatically derives test cases from the formally specified requirements. By generating these tests, FRET helps ensure that critical system behaviors described in the requirements are adequately exercised, promoting early detection of specification errors and supporting rigorous system verification. This integration of test generation reinforces the role of FRET as a comprehensive tool for formal requirements engineering and validation. To show an example of the Test Case Generation features, it was selected a component that had 2 requirements associated to it. In figure 4.12 shows how the test case generation tab is presented after doing a successful generation of test cases for the Airbiquity_server component. It shows to the user the state of completion, how many test cases were generated and the time it took to create them.

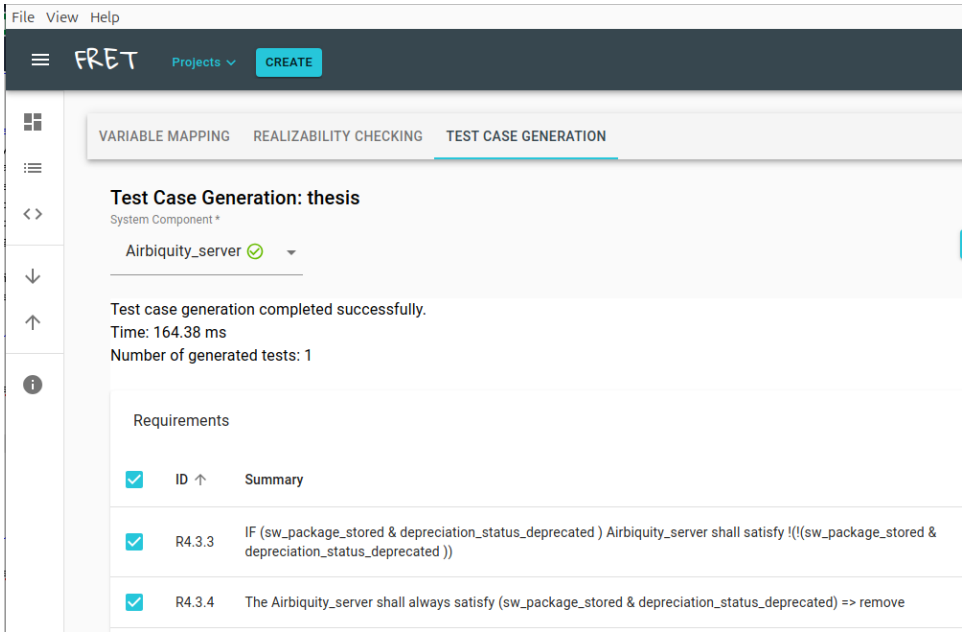


Figure 4.12: Successful Test Case Generation

After generating test cases, FRET allows users to simulate them and to export them to other tools. The figures 4.13 and 4.14 shows how the created test case can be studied in a simulation using LTLsim. Both figures are related to the same test case.

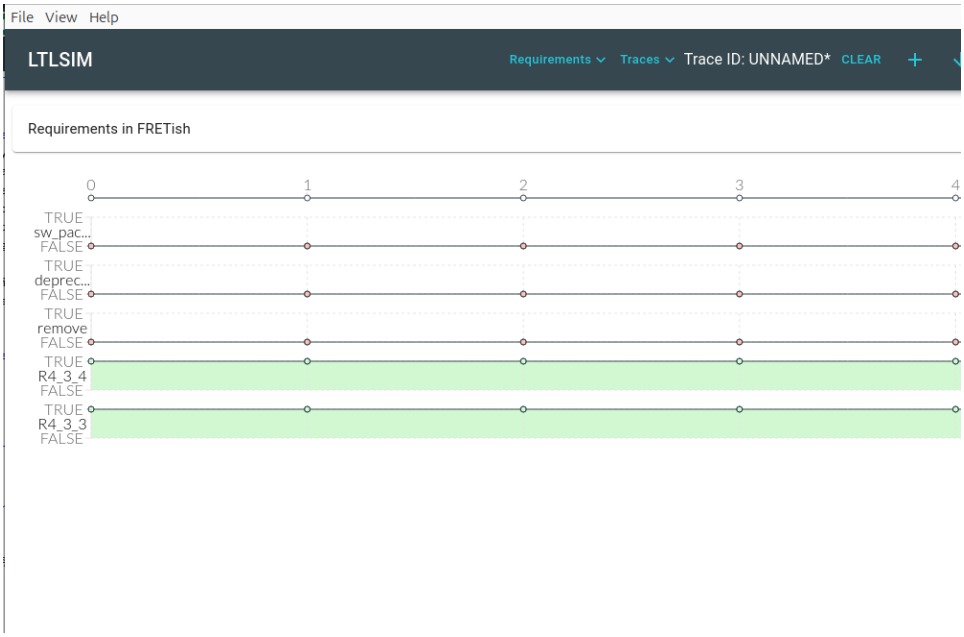


Figure 4.13: Successful Test Case Simulation

The unsuccessful simulation result from figure 4.14 was achieved from the user changing the states of the variables sw_package_stored and depreciation_status_deprecated on the second trace point of the simulation.

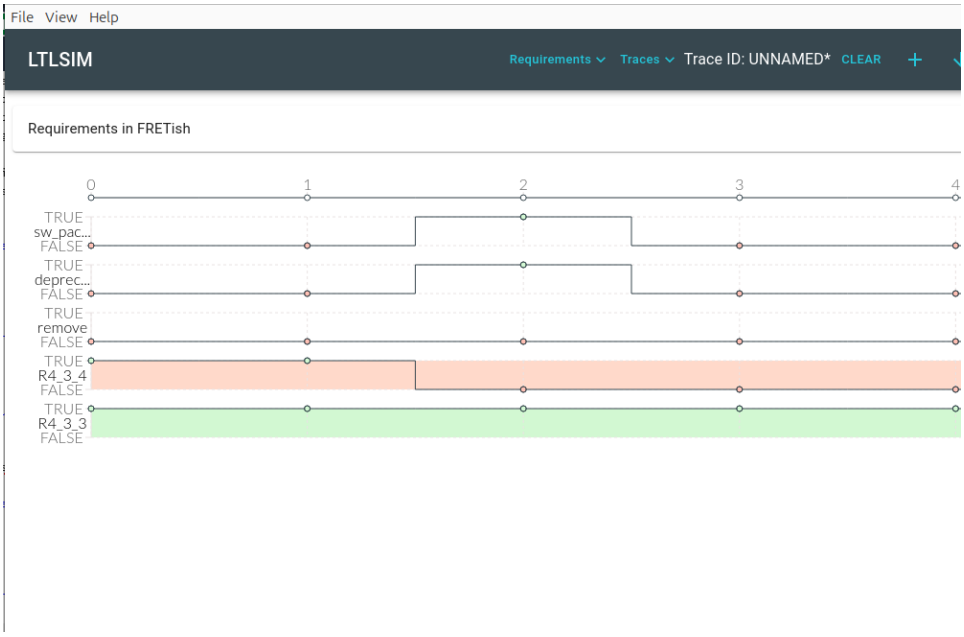


Figure 4.14: Unsuccessful Test Case Simulation

Since FRET is a tool that has the possibility of integration with several different external tools, the generated test cases are also possible to export into a .json format to use in an external tool. Figure 4.15 shows the output from the exportation of the example generated test case.

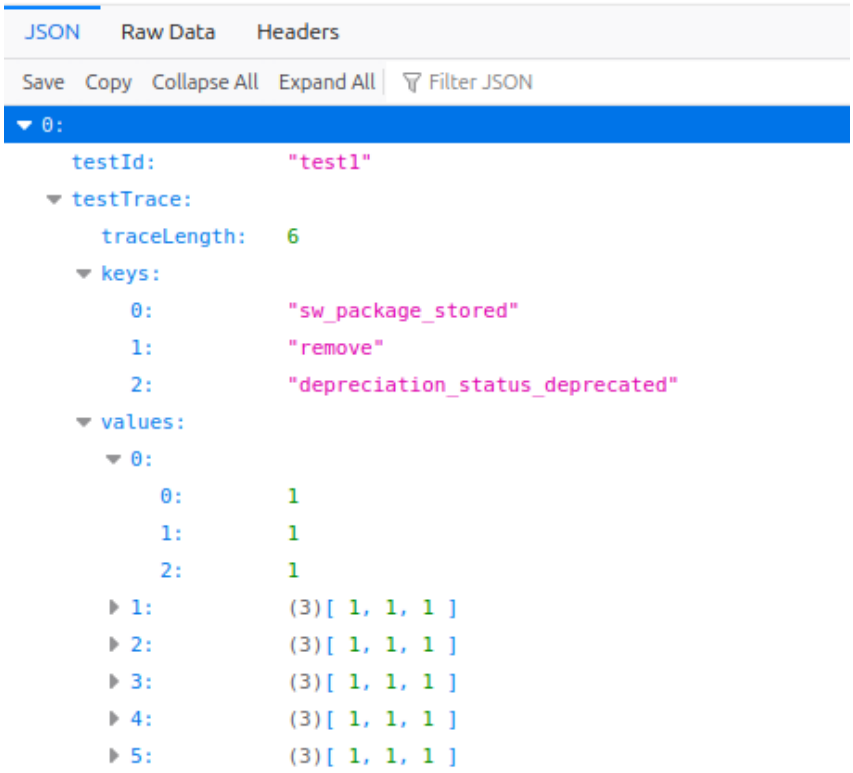


Figure 4.15: Exportation of generated test case

This .json file , has the name of the test, the number of traces used in the temporal simulations and the values in them. This example in particular, had all variables to be booleans with the "true" attribution, hence showing all variables values to '1'.

This exploration of FRET's test case generation capabilities demonstrates the tool's potential to bridge formal requirements and practical validation. The core of this comprehensive investigation was to understand and experiment with alternatives on requirement analysis. Beyond the initial selection, each requirement was carefully formalized, assessed for completeness, and subjected to rigorous clarification. This process ensured that ambiguities and omissions typical of natural language specifications were systematically resolved, helping establish a robust foundation for formal verification. The comparison between the EARS and FRET approaches and the evaluation of their results side-by-side, reveals important distinctions in expressiveness, usability, and analytical depth. EARS enables rapid and structured elicitation with clear templates, but its scope is limited compared to FRET's extensive support for formal verification and nuanced variable mapping. FRET's strengths are most visible when dealing with complex requirements, enabling deeper consistency checking. The in-depth analysis, and the comparison of specification approaches contribute to a more rigorous and reflective methodology, enhancing both the reliability and clarity of the system requirements.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The use of formal requirements specification languages offers significant advantages, as demonstrated by the experience in this thesis. Employing formal languages demands rigorous attention to the identification and precise definition of requirements, which substantially contributes to clarifying ambiguities and ensuring a comprehensive understanding of project necessities.

Although the FRET tool provides valuable support for formal requirements elicitation and analysis, its usability challenges and insufficient documentation create barriers to more effective and widespread adoption. Nevertheless, it is clear from this research that FRET already offers meaningful benefits, particularly for projects centered on high-level functional requirements, where its framework facilitates thorough and systematic requirements management.

This research successfully explored FRET in considerable detail and applied it to a representative case study with relative success. However, improved and more extensive documentation from FRET would have enabled a deeper and more detailed examination of the tool full capabilities and potential limitations.

Ultimately, the primary objective of this thesis was achieved, obtaining a thorough understanding on how formal requirements tools such as FRET operate and their potential impact on formal requirements engineering. This work lays a foundation for future studies and enhancements aimed at increasing the usability and applicability of formal requirements tools in practical engineering contexts.

5.2 Future Work

Future work for this research focus on enhancing the usability of the FRET tool to make formal requirements elicitation more accessible and efficient for practitioners. One promising direction is integrating natural language processing techniques to automate the translation pipeline from natural language requirements to formal specifications. Specifically, automating the conversion from natural language to EARS, and subsequently from EARS to FRETish, would significantly streamline the methodology by reducing manual effort and minimizing the risk of human error.

To advance this automation, and improve translation accuracy and quality, the recent concept of agentic artificial intelligence (AI) could be leveraged. Agentic AI, which involves multiple interacting agents deliberating and refining intermediate results collaboratively, can be employed to enhance translation accuracy and consistency between natural language,

EARS, and FRETish formats. This would enhance the reliability and consistency in the automated translation from natural language to formal specifications within FRET.

Another avenue for future exploration involves training a small language model (SLM) specifically tailored for this translation task. Developing an open-source SLM for translating natural language to EARS and from EARS to FRETish would encourage community engagement and foster collaborative improvements. This approach would not only benefit academic research but also support smaller commercial teams and startups that could adopt this methodology for compact projects without extensive formal methods expertise, lowering the barrier to entry for rigorous requirements engineering.

By pursuing these enhancements, it would substantially advance the accessibility and scalability of formal requirements elicitation tools, contributing to more widespread adoption of more reliable and secure systems.

Bibliography

[1] *NASA-SW-VnV/fret: A framework for the elicitation, specification, formalization and analysis of requirements*. https://github.com/NASA-SW-VnV/fret. Accessed 2025-09-08. 2025.

[2] NASA Ames Research Center. *Formal Requirements Elicitation Tool (FRET)*. https://software.nasa.gov/software/ARC-18066-1. Accessed: 2025-09-09. 2022.

[3] "Standardization and technical standards". In: (2019). URL: https://www.vda.de/en/topics/automotive-industry/standardization-and-technical-standards.

[4] *Requirements engineering for the automotive industry*. Tech. rep. IBM Rational, 2009. URL: https://public.dhe.ibm.com/software/emea/de/rational/neu/Requirements_Engineering_for_the_automotive_industry_EN_2009.pdf.

[5] "Automotive Standards: The Comprehensive Guide". In: (2024). URL: https://compliance-aspekte.de/en/articles/checklist-of-mandatory-standards-for-automotive-industry/.

[6] "Automotive Functional Safety: Ensuring ISO 26262 Compliance". In: (2025). URL: https://www.apriorit.com/dev-blog/automotive-functional-safety-ensuring-iso-26262-compliance.

[7] "Automotive Standards". In: (2023). URL: https://www.intertekinform.com/en-gb/key-standards/automotive-standards/.

[8] "Why Quality Management (QM) is required for easing ISO 26262 activities". In: (2025). URL: https://www.quidditytech.io/post/why-quality-management-qm-is-required-for-easing-iso-26262-2018-activities.

[9] *A Step-By-Step Automotive Requirements Management Plan*. 2025. URL: https://www.requiment.com/automotive-requirements-management-plan/.

[10] Schmal2024. "Leveraging Natural Language Processing for a Consistency Checking Toolchain of Automotive Requirements". In: (2024). URL: https://www.se-rwth.de/publications/Leveraging-Natural-Language-Processing-for-a-Consistency-Checking-Toolchain-of-Automotive-Requirements.pdf.

[11] "Local large language models to simplify requirement engineering". In: *Journal of Software Engineering* (2023). URL: https://www.tandfonline.com/doi/full/10.1080/21693277.2024.2375296.

[12] *Natural Language Processing for Requirements Engineering*. 2023. URL: https://upcommons.upc.edu/bitstreams/f5211837-9627-4e93-a101-9cb8deca115c/download.

[13] James Howden, Leandros Maglaras, and Mohamed Amine Ferrag. "The Security Aspects of Automotive Over-the-Air Updates". In: *International Journal of Cyber Warfare and Terrorism* 10.2 (2020), pp. 64–81. DOI: 10.4018/IJCWT.2020040104. URL: http://services.igi-global.com/resolvedoi/resolve.aspx?doi=10.4018/IJCWT.2020040104.

[14] *Computer-aided Formal Specification to enable a fully automated requirements analysis*. Tech. rep. BTC Embedded Systems, 2024. URL: https://www.btc-embedded.

com/downloads/whitepapers/computer-aided-formal-specification-to-enable.pdf.

[15] Xinchi He et al. "Securing Over-The-Air IoT Firmware Updates using Blockchain". In: *Proceedings of the International Conference on Omni-Layer Intelligent Systems*. Crete, Greece: ACM, 2019, pp. 164–171. DOI: 10.1145/3312614.3312649. URL: https://dl.acm.org/doi/10.1145/3312614.3312649.

[16] Amrita Ghosal, Subir Halder, and Mauro Conti. "STRIDE: Scalable and Secure Over-The-Air Software Update Scheme for Autonomous Vehicles". en. In: *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*. Dublin, Ireland: IEEE, June 2020, pp. 1–6. ISBN: 978-1-72815-089-5. DOI: 10.1109/ICC40277.2020.9148649. URL: https://ieeexplore.ieee.org/document/9148649/ (visited on 01/13/2023).

[17] Francesco Tango, Giuseppe Tanganelli, and Michele G. C. A. Cimato. "A Systematic Review of Over-the-Air Update Approaches in Automotive Systems". In: *IEEE Access* 9 (2021), pp. 102459–102476.

[18] Ahmad Al-Bataineh, Giandomenico Spezzano, and Paolo Brizzi. "Over-the-Air Software Updates for Connected Vehicles: Security and Performance Challenges". In: *Vehicular Communications* 33 (2022), p. 100417.

[19] Ian Sommerville. *Software Engineering*. 9th. Addison-Wesley, 2011.

[20] Elizabeth Hull, Ken Jackson, and Jeremy Dick. *Requirements Engineering*. 4th. Springer, 2017.

[21] Klaus Pohl. *Requirements Engineering: Fundamentals, Principles, and Techniques*. Springer, 2010.

[22] Martin Glinz. "A Glossary of Requirements Engineering Terminology". In: *Requirements Engineering Magazine* (2013).

[23] Uwe Winkelhake. *The Digital Transformation of the Automotive Industry*. Cham: Springer International Publishing, 2018. DOI: 10.1007/978-3-319-71610-7. URL: http://link.springer.com/10.1007/978-3-319-71610-7.

[24] Taehyoung Kim and Sungkwon Park. "Compare of Vehicle Management over the Air and On-Board Diagnostics". In: *2019 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS)*. IEEE, 2019, pp. 1–2. DOI: 10.1109/ISPACS48206.2019.8986260. URL: https://ieeexplore.ieee.org/document/8986260/.

[25] Houssem Guissouma, Axel Diewald, and Eric Sax. "A Generic System for Automotive Software Over the Air (SOTA) Updates Allowing Efficient Variant and Release Management". In: *Information Systems Architecture and Technology: Proceedings of 39th International Conference on Information Systems Architecture and Technology – ISAT 2018*. Ed. by Leszek Borzemski, Jerzy Świątek, and Zofia Wilimowska. Springer International Publishing, 2019, pp. 78–89. DOI: 10.1007/978-3-319-99981-4_8. URL: http://link.springer.com/10.1007/978-3-319-99981-4_8.

[26] Thilo Streichert and Matthias Traub. *Elektrik/Elektronik-Architekturen im Kraftfahrzeug*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. DOI: 10.1007/978-3-642-25478-9. URL: http://link.springer.com/10.1007/978-3-642-25478-9.

[27] Subir Halder, Amrita Ghosal, and Mauro Conti. "Secure OTA Software Updates in Connected Vehicles: A Survey". In: *arXiv preprint* (2019). arXiv:1904.00685 [cs]. URL: http://arxiv.org/abs/1904.00685.

[28] Houssem Guissouma, Axel Diewald, and Eric Sax. "A Generic System for Automotive Software Over the Air (SOTA) Updates Allowing Efficient Variant and Release Management". In: *Advances in Intelligent Systems and Computing* 852 (2019), pp. 78–89. DOI: 10.1007/978-3-319-99981-4_8.

[29] Thomas Chowdhury et al. "Safe and Secure Automotive Over-the-Air Updates". In: *Developments in Language Theory*. Vol. 11088. Springer International Publishing, 2018, pp. 172–187. DOI: 10.1007/978-3-319-99130-6_12. URL: http://link.springer.com/10.1007/978-3-319-99130-6_12.

[30] European Union. *Regulation (EU) 2016/679 (General Data Protection Regulation)*. https://eur-lex.europa.eu/eli/reg/2016/679/oj. Accessed 2025-09-04. 2016.

[31] UNECE WP.29. *UN Regulation No. 155 - Cyber Security and Cyber Security Management System*. https://unece.org/transport/documents/2020/12/wp29-regulations-cyber-security-and-cyber-security-management. Accessed 2025-09-04. 2020.

[32] UNECE WP.29. *UN Regulation No. 156 - Software Update and Management System*. https://unece.org/transport/documents/2020/12/wp29-regulations-software-update-vehicle. Accessed 2025-09-04. 2020.

[33] Thomas Strang et al. "Secure Automotive On-Board Protocols: A Case of Over-the-Air Firmware Updates". In: *Communication Technologies for Vehicles*. Springer Berlin Heidelberg, 2011, pp. 224–238. DOI: 10.1007/978-3-642-19786-4_20. URL: http://link.springer.com/10.1007/978-3-642-19786-4_20.

[34] Byungjoo Kim and Sungkwon Park. "ECU Software Updating Scenario Using OTA Technology through Mobile Communication Network". In: *2018 IEEE 3rd International Conference on Communication and Information Systems (ICCIS)*. 2018, pp. 67–72. DOI: 10.1109/ICOMIS.2018.8645019. URL: https://ieeexplore.ieee.org/document/8645019/.

[35] Subir Halder, Amrita Ghosal, and Mauro Conti. "Secure over-the-air software updates in connected vehicles: A survey". en. In: *Computer Networks* 178 (2020), p. 107343. ISSN: 13891286. DOI: 10.1016/j.comnet.2020.107343. URL: https://linkinghub.elsevier.com/retrieve/pii/S1389128619314963 (visited on 01/18/2023).

[36] Muzaffar Khurram et al. "Enhancing connected car adoption: Security and over the air update framework". In: *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*. Reston, VA, USA: IEEE, 2016, pp. 194–198. DOI: 10.1109/WF-IoT.2016.7845430. URL: http://ieeexplore.ieee.org/document/7845430/.

[37] Jonas Römer. "Software Updates Over-The-Air II". English. In: ().

[38] D. K. Nilsson and U. E. Larson. "Secure Firmware Updates over the Air in Intelligent Vehicles". In: *ICC Workshops - 2008 IEEE International Conference on Communications Workshops*. Beijing, China: IEEE, 2008, pp. 380–384. DOI: 10.1109/ICCW.2008.78. URL: http://ieeexplore.ieee.org/document/4531926/.

[39] Uptane Alliance. *Uptane: The secure software update framework for automobiles*. https://uptane.org. Accessed: 2025-09-04.

[40] Uptane Alliance. *Uptane Specification v2.1*. https://uptane.org/docs/2.1.0/standard. Accessed: 2025-09-04. 2019.

[41] Daniel Larraz and et al. "A Comprehensive, Automated Security Analysis of the Uptane Framework". In: *Proceedings of the ACM on Computer and Communications Security* (2024). Preprint available online.

[42] *Uptane - Linux Foundation Wiki*. https://wiki.linuxfoundation.org/gsoc/uptane. Accessed: 2025-09-04.

[43] Subir Halder, Amrita Ghosal, and Mauro Conti. "Secure over-the-air software updates in connected vehicles: a survey". In: *Computer Networks* 178 (2020), p. 107343. DOI: 10.1016/j.comnet.2020.107343.

[44] *Airbiquity integrates Kaspersky platform for secure OTA updates*. https://www.kaspersky.com/news/airbiquity-integrates-kaspersky-platform-for-secure-ota-updates. Accessed: 2025-09-04. 2022.

[45] A. O. J. Sabriye and W. M. N. W. Zainon. "A Framework for Detecting Ambiguity in Software Requirement Specification". In: *8th International Conference on Software Engineering and Computer Systems*. 2017.

[46] P. Achimugu et al. "A systematic literature review of software requirements prioritization research". In: *Information and Software Technology* 56 (6 2014), pp. 568–590.

[47] Philipp Reinkemeier et al. "A pattern-based requirement specification language: Mapping automotive specific timing requirements". In: ().

[48] Andy Mavin et al. "Easy Approach to Requirements Syntax (EARS)". In: *17th IEEE International Requirements Engineering Conference (RE)*. 2009, pp. 317–322. DOI: 10.1109/RE.2009.53.

[49] McKinsey & Company. *Ford Mustang Mach-E Recall Driven by Firmware Integration Challenges*. https://www.electraytech.com/why-automotive-software-fails-10-deep-rooted-issues-slowing-progress-on-the-road/. Accessed Sept 2025. 2022.

[50] McKinsey & Company. *Volkswagen ID.3 Launch Delay Due to Software Integration Problems*. https://www.electraytech.com/why-automotive-software-fails-10-deep-rooted-issues-slowing-progress-on-the-road/. Accessed Sept 2025. 2020.

[51] Jama Software. *Adopting EARS Notation for Requirements Engineering*. Accessed: 2025-09-09. 2025. URL: https://www.jamasoftware.com/requirements-management-guide/writing-requirements/adopting-the-ears-notation-to-improve-requirements-engineering/.

[52] Irem Aktug and Katsiaryna Naliuka. "ConSpec — A formal language for policy specification". In: *Science of Computer Programming* 74.1-2 (2008), pp. 2–12. DOI: 10.1016/j.scico.2008.09.004. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167642308001056.

[53] Anastasia Mavridou et al. "Formal Requirements Analysis with FRET: A Tool Overview". In: *Software Tools for Technology Transfer* (2022).

[54] IBM Corporation. *IBM Rational DOORS Documentation*. 2019. URL: https://www.ibm.com/docs/en/ermd/9.7?topic=requirements-rational-doors.

[55] DOORSTOP Development Team. *DOORSTOP: Requirements Authoring Tool*. 2025. URL: https://github.com/doorstop-tool/doorstop.

[56] *Requirements engineering for the automotive industry*. Tech. rep. IBM Rational, 2009. URL: https://public.dhe.ibm.com/software/emea/rational/neu/RequirementsEngineeringfortheautomotiveindustryEN2009.pdf.

[57] Andreas Katis et al. "Capture, Analyze, Diagnose Realizability Checking Of Requirements in FRET". In: *Computer Aided Verification*. Springer International Publishing, 2022. DOI: 10.1007/978-3-031-13188-2_24.

[58] *Doorstop: Open Source Requirements Management Tool*. https://doorstop.readthedocs.io. Accessed 2025-09-07.

[59] *doorstop-edit: Graphical User Interface for Doorstop*. https://github.com/eerimoq/doorstop-edit. Accessed 2025-09-07.

[60] Dimitra Giannakopoulou et al. "Formal Requirements Elicitation with FRET". In: ().

[61] NASA Formal Requirements Elicitation Tool Development Team. *FRET User Manual*. https://github.com/NASA-SW-VnV/fret. Accessed 2025-09-09.

[62] Adrien Champion et al. "CoCoSpec: A Mode-aware Contract Language for Reactive Systems". In: *NASA Formal Methods Symposium (NFM)*. 2016, pp. 58–74.

[63] Shivendra Bhattacharyya et al. "Contract-Based Compositional Verification of Safety-Critical Embedded Systems". In: *Proceedings of the 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2024, pp. 341–346.

[64] *CoPilot Runtime Verification Tool*. https://www.aquamonitorgroup.com/copilot/. Accessed 2025-09-09.

[65] Dimitra Giannakopoulou et al. *Realizability Checking of Requirements in FRET*. Tech. rep. NASA Technical Reports, 2022. URL: https://ntrs.nasa.gov/api/citations/20220007510/downloads/TechnicalReport__FRET_Realizability_Checking.pdf.

[66] Andreas Katis et al. "Capture, Analyze, Diagnose Realizability Checking Of Requirements in FRET". In: (2022). DOI: 10.1007/978-3-031-13188-2_24.

[67] Andrew Gacek, John Backes, and Darren Cofer. *JKind Model Checker*. https://github.com/agacek/jkind. Accessed 2025-09-08.

[68] Mickael Champion, Thanh-Hai Nguyen, and Corina S. Pasareanu. *Kind 2 Model Checker*. https://github.com/kind2-mc/kind2. Accessed 2025-09-08.

[69] Andreas Katis and Author Anaísa. *JKind vs Kind2 and MBP*. https://github.com/NASA-SW-VnV/fret/discussions/116. GitHub discussion, accessed 2025-09-08. 2025.

[70] Amir Pnueli. *The Temporal Logic of Programs*. 1977, pp. 46–57.

[71] Alessandro Cimatti et al. "NuSMV: A New Symbolic Model Checker". In: *International Journal on Software Tools for Technology Transfer* 2.4 (2000), pp. 410–425. DOI: 10.1007/s100090050021. URL: https://link.springer.com/article/10.1007/s100090050021.

[72] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. "Model Checking". In: (1999).

Appendix A

Software installation guidelines

A.1 Software Installation

A.1.1 Introduction

The FRET software, it is a package necessary to be installed in the machine where you want to use it. It can not be used in browser mode. This software also has some dependencies that are necessary to install in order to work. FRET is recommended to be used in a Linux environment, but is also possible to exist in Windows Operating Systems and in macOS ones. Since the recommended OS to use FRET is Linux, this recommendation was followed on this thesis work. In any of the OS environments it is necessary to use an account with admin privileges for all the required installations. This guide is valid for Linux Ubuntu 24.04.2 LTS version and has an installation time of around 2 to 3 hours.

A.1.2 Dependencies

Linux packages

It is necessary to install the curl package, as it provides a command-line tool for data transfer using various network protocols. In practice, curl will be used to retrieve files or web resources directly from the terminal, which facilitates automated package installation and dependency management.

```
sudo apt update
sudo apt install curl
sudo apt install libboost-all-dev
sudo apt install libgtk-3-0t64 libdrm2 libgbm1 libnss3 libx11-xcb1
    libasound2t64
sudo apt install libgmp-ddev
sudo apt install bison
```

NodeJS

Node.js is required as a runtime dependency for FRET , ensuring the execution of JavaScript-based components and their associated packages.

```
# Download and install nvm:
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.2/install.sh
     | bash
# Restart the shell to take effect
```

```
\. "$HOME/.nvm/nvm.sh"
# Download and install Node.js:
nvm install 20
# Verify the Node.js version:
node -v # Should print "v20.19.0".
nvm current # Should print "v20.19.0".
# Verify npm version:
npm -v # Should print "10.8.2".
```

Python

To run FRET is necessary to use a python between v3.10.x to v3.13.x. In this guide it was used Python 3.13.4.

```
sudo apt install python3.13
sudo apt install python3-setuptools
sudo apt install python3-apt
```

Make sure, that the call to Python 3.13.4 is "python" and not "python3" otherwise, some calls will not work.

```
sudo apt install python-is-python3
```

Git

Git is necessary to be installed and it can be any version.

```
sudo apt install git
```

Compilers

It is necessary to install GCC, G++, Make, and CMake, as they provide the compilers and build tools required for compiling and building FRET. These tools ensure proper handling of source code translation, dependency management, and project configuration.

```
sudo apt install gcc
sudo apt install g++
sudo apt install make
sudo apt install cmake
```

NuSMV

NuSMV is used by FRET as a model checker to verify formalized requirements by analyzing system states and ensuring specification correctness during the verification process. It is distributed as pre-compiled binaries. Download the checker from the link below:

```
https://nusmv.fbk.eu/distrib/2.7.0/NuSMV-2.7.0-linux64.tar.xz
```

Then is just extract the archive to the Home directory, and add to the system PATH to enable direct execution from the command line. To test that is working just call on the command line "NuSVM" and it should show up lie:

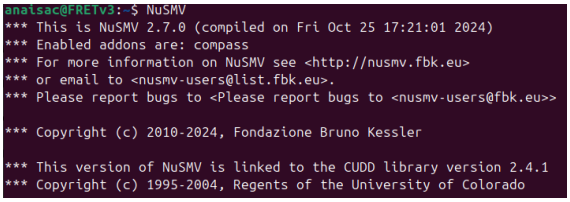


Figure A.1: Confirmation of NuSMV

Z3

Z3 is a theorem prover used by FRET to automatically check logical consistency and satisfiability of formalized requirements, supporting rigorous verification of system properties during analysis. It is necessary to download it from:

```
https://github.com/Z3Prover/z3/archive/refs/tags/z3-4.14.1.tar.gz
```

Then extract it into the Home directory. After it, open a command line inside the extracted z3-z3-4.14.1 folder and run the following commands:

```
python scripts/mk_make.py
cd build
make
sudo make install
```bash
```

This installation takes a while in the "make" step To test the correct installation run "z3 -h" and you should see this:

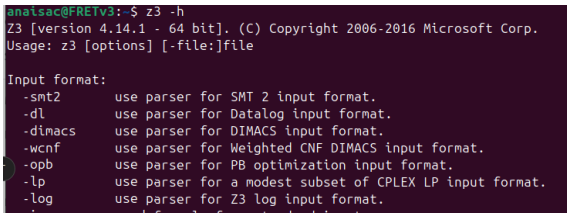


Figure A.2: Z3 test with success

Java

For JKind is necessary to have at least JAVA 8 installed.
```

```
sudo apt install default-jre
java -version
```

JKind

JKind is an infinite-state model checker integrated with FRET to perform realizability checking of formalized requirements, enabling verification of system properties and diagnosis of unrealizable specifications. JKind can be download it from:

```
https://github.com/andreaskatis/jkind-1/releases/tag/v2.2
```

Confirm you have full permissions for executing Jkind scripts. If not

```
ls -l
chmod -R +x jkind/
```

Then extract it into the Home directory and add it to the system PATH. To test the correct installation run "jkind" and you should see this:

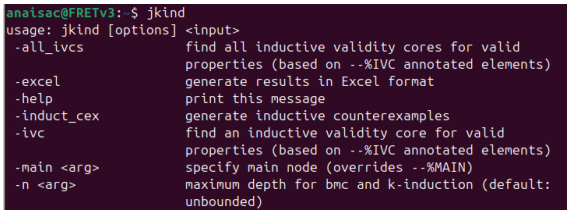


Figure A.3: JKind test with success

Kind2

Kind2 is a model checker integrated with FRET to verify Lustre code and formalized requirements, enabling automated consistency and correctness checking of system models during the verification process. The version in use was v2.3.0. Kind2 can be download it from:

```
https://kind2-mc.github.io/kind2/
```

Then extract it into the Home directory and add it to the system PATH. To test the correct installation run "kind2 -h" and you should see this:

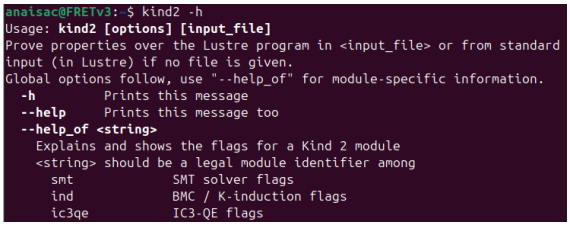


Figure A.4: Kind2 test with success

AEVAL

AEVAL is a formal verification tool used by FRET to evaluate and verify temporal logic properties derived from requirements, supporting rigorous analysis of system behavior against specified constraints. Get AEVAL from the following repository:

```
https://github.com/grigoryfedyukovich/aeval
```

After getting the repository in the Home directory do the following commands:

```
cd aeval
mkdir build
cd build
cmake ../
make
```

At the end add the path to the aeval script to the system PATH. To test the correct installation run "aeval –help" and you should see this:

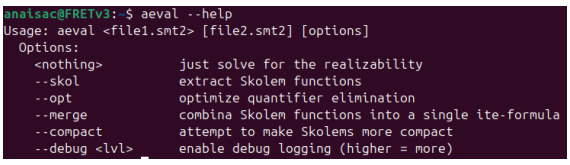


Figure A.5: AEval test with success

PATHs

Before proceeding to the FRET installation, make sure you have all of these tools in the PATH of your machine:

Figure A.6: Confirmation of PATHs

A.1.3 FRET installation instructions

The FRET tool is found in the following repository:

```
https://github.com/NASA-SW-VnV/fret
```

After all the previous dependencies are installed, to install the tool follow the next steps:

```
git clone https://github.com/NASA-SW-VnV/fret.git
cd fret/fret-electron
npm run fret-install
```

FRET also uses an chrome based engine for the GUI so it is necessary to give additional permissions to run:

```
# Navigate to the electron directory
cd ~/fret/fret-electron/node_modules/electron/dist
# Set ownership and permissions for chrome-sandbox
sudo chown root:root chrome-sandbox
sudo chmod 4755 chrome-sandbox
```

To run the LTL simulator, is necessary to add this directory to your system path:

```
PATH_TO_FRET/fret/tools/LTLSIM/ltlsim-core/simulator
```

To enable unprivileged users to create isolated namespaces, which is necessary for container and rootless software functionality it's necessary:

```
sudo sysctl kernel.unprivileged_userns_clone=1
```

To start FRET go to the directory:

```
cd ~/fret/fret-electron/
npm start
```

Following these steps will ensure the software is correctly installed and configured.