NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

AURÉLIO MIRANDA SANTOS RODRIGUES GABOLEIRO

BSc in Computer Science and Engineering

RAMSES: A CONFIGURATION LANGUAGE FOR AUTOMATIC CODE GENERATION IN AN INDUSTRIAL CONTEXT.

A DOMAIN-SPECIFIC LANGUAGE APPROACH FOR SAFETY-CRITICAL EMBEDDED SYSTEMS

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
Draft: *December 3, 2025*

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

RAMSES: A CONFIGURATION LANGUAGE FOR AUTOMATIC CODE GENERATION IN AN INDUSTRIAL CONTEXT.

A DOMAIN-SPECIFIC LANGUAGE APPROACH FOR SAFETY-CRITICAL EMBEDDED SYSTEMS

AURÉLIO MIRANDA SANTOS RODRIGUES GABOLEIRO

BSc in Computer Science and Engineering

Adviser: Miguel Goulão
Associate Professor, NOVA-LINCS, NOVA School of Science and Techonology

Co-adviser: Dominique Blouin
Associate Professor, Télécom Paris, Institut Polytechnique de Paris

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
Draft: *December 3, 2025*

Abstract

The automatic generation of code from templates is a widely adopted approach in the industry to reduce costs and increase software reliability. However, this generation has to be highly configurable to meet specific requirements, such as project coding practices, compatibility with APIs and performance optimizations.

RAMSES is an Architecture Analysis and Design Language (AADL) code generation tool that fully automates the process of converting AADL models into code to support the design of embedded and cyber-physical systems. The most significant advantage of RAMSES is its ability to automatically generate code from high-level models, eliminating implementation details and providing better portability and reusability. However, as industrial systems become increasingly diverse, the need to adapt to specific industrial environments requires an adaptable configuration of the generated code.

This project proposes the design and implementation of a configuration language for RAMSES, enabling code generation to be customized according to specific requirements of each industry. The steps involve defining the syntax and semantics of the language, integrating it into RAMSES and testing it through industrial scenarios.

Throughout the project, various scenarios will be considered to demonstrate the effectiveness of the solution in comparison with other tools in this context. The aim is to provide an intuitive and helpful tool that can be used to make the language adaptable.

Keywords: Code Generation, AADL, RAMSES, Industrial Automation

Resumo

A geração automática de código a partir de modelos é uma abordagem amplamente adoptada na indústria para reduzir custos e aumentar a fiabilidade do software. No entanto, esta geração tem de ser altamente configurável para responder a requisitos específicos, tais como práticas de codificação de projectos, compatibilidade com APIs e optimizações de desempenho.

O RAMSES é uma ferramenta de geração de código para Architecture Analysis and Design Language (AADL) que automatiza totalmente o processo de conversão de modelos AADL em código para apoiar a conceção de sistemas integrados e ciber-físicos. A vantagem mais significativa do RAMSES é a sua capacidade de gerar automaticamente código a partir de modelos de alto nível, eliminando os pormenores de implementação e proporcionando uma melhor portabilidade e reutilização. No entanto, à medida que os sistemas industriais se tornam cada vez mais diversificados, a necessidade de adaptação a ambientes industriais específicos requer uma configuração adaptável do código gerado.

Este projeto propõe a conceção e implementação de uma linguagem de configuração para o RAMSES, permitindo que a geração de código seja personalizada de acordo com os requisitos específicos de cada indústria. As etapas envoluem a definição da sintaxe e semântica da linguagem, a sua integração no RAMSES e o seu teste através de cenários industriais.

Ao longo do projeto, serão considerados vários cenários para demonstrar a eficácia da solução em comparação com outras ferramentas neste contexto. O objetivo é fornecer uma ferramenta intuitiva e útil que possa ser utilizada para tornar a linguagem adaptável.

Palavras-chave: Geração de Código, AADL, RAMSES, Automação Industrial

Résumé

La génération automatique de code à partir de modèles est une approche largement adoptée dans l'industrie pour réduire les coûts et augmenter la fiabilité des logiciels. Cependant, cette génération doit être hautement configurable pour répondre à des exigences spécifiques, telles que les pratiques de codage du projet, la compatibilité avec les API et les optimisations de performance.

RAMSES est un outil de génération de code de Architecture Analysis and Design Language (AADL) qui automatise entièrement le processus de conversion des modèles AADL en code pour soutenir la conception de systèmes embarqués et cyber-physiques. L'avantage le plus important de RAMSES est sa capacité à générer automatiquement du code à partir de modèles de haut niveau, en éliminant les détails d'implémentation et en offrant une meilleure portabilité et réutilisation. Cependant, les systèmes industriels devenant de plus en plus diversifiés, la nécessité de s'adapter à des environnements industriels spécifiques requiert une configuration adaptable du code généré.

Ce projet propose la conception et la mise en œuvre d'un langage de configuration pour RAMSES, permettant de personnaliser la génération de code en fonction des exigences spécifiques de chaque industrie. Les étapes consistent à définir la syntaxe et la sémantique du langage, à l'intégrer dans RAMSES et à le tester à l'aide de scénarios industriels.

Tout au long du projet, divers scénarios seront envisagés pour démontrer l'efficacité de la solution par rapport à d'autres outils dans ce contexte. L'objectif est de fournir un outil intuitif et utile qui peut être utilisé pour rendre le langage adaptable.

Mots-clés : Génération de code, AADL, RAMSES, Automatisation industrielle

Contents

List of Figures

Glossary

API Application programming interface (API) is a connection between computers or between computer programs. It is a type of software interface, offering a service to other pieces of software. *(pp. 2, 3, 15)*

AADL Architecture Analysis and Design Language is a textual and graphical language that can be used to design and analyze the software and hardware architecture of performance-critical real-time systems. *(pp. 1–6, 8, 13, 14, 17, 18, 20, 21, 23, 25, 31, 35, 39, 45, 46, 66, 68, 70)*

CPS Cyber-physical systems are mechanisms controlled and monitored by computer algorithms, tightly integrated with the internet and its users. In cyber-physical systems, physical and software components are deeply intertwined, able to operate on different spatial and temporal scales, exhibit multiple and distinct behavioral modalities, and interact with each other in ways that change with context. *(pp. 1, 2)*

DSL Domain-specific language (DSL) is a computer language specialized to a particular application domain. Its created specifically to solve problems in a particular domain and is not intended to be able to solve problems outside of it (although that may be technically possible). *(pp. 3, 16, 18, 22, 25, 40, 43, 46, 48–51, 65–69, 72)*

Identifier A symbolic name used in programming languages to uniquely designate elements such as variables, functions, classes, or types. Identifiers allow programmers to reference and manipulate these entities in code. Their form is subject to the syntactic rules of the language (e.g., starting with a letter or underscore, excluding reserved keywords). *(pp. 7, 17–21, 27–29, 31, 51, 59, 60, 77)*

JSON JavaScript Object Notation is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. *(pp. 32–34, 44, 60)*

JUnit JUnit is a test automation framework for the Java programming language. JUnit is often used for unit testing, and is one of the xUnit frameworks. *(pp. 44, 65)*

metamodel A model that defines the structure, semantics, and constraints of other models. In model-driven engineering, a metamodel specifies the types of elements and relationships that can appear in a model, effectively defining the modeling language. For example, an UML metamodel defines what a "Class", "Attribute" or "Operation" is and how they relate. *(p. 21)*

M2T Model-to-Text (M2T) is a form of model transformation where structured models are automatically converted into textual artifacts, such as source code, configuration files, or documentation. *(pp. 5, 9, 59)*

XML XML markup language and file format for storing, transmitting, and reconstructing data. It defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. *(p. 63)*

Acronyms

AUTOSAR AUTomotive Open System ARchitecture *(p. 9)*

CSS Cascading Style Sheets *(p. 34)*

DI Department of Computer Science *(p. 2)*
DSML Domain Specific Modeling Language *(p. 18)*

EMF Eclipse Modeling Framework *(pp. 9, 23)*

GQM Goal Question Metric *(p. 66)*

HTML HyperText Markup Language *(pp. 34, 35, 43, 60)*

IEC International Electrotechnical Commission Standard *(p. 9)*
ISO International Organization for Standardization *(pp. 15, 67)*

MBE Model-Based Engineering *(pp. 4, 5)*
MD Model-Driven *(pp. 1, 10)*
MISRA Motor Industry Software Reliability Association *(pp. 9, 10, 12, 13, 15)*

OS Operating System *(pp. 39, 43)*

RAMSES Refinement of AADL Models for the Synthesis of Embedded Systems *(pp. 1–3, 5–13, 17, 18, 20, 23, 25, 27, 30, 31, 36, 39, 44, 46, 48, 49, 51, 58, 61, 62, 66, 67, 69)*
ROS Robot Operating System *(pp. 27, 28, 30, 35–39, 44, 49, 53, 59, 64, 66–68, 70, 72, 77)*

SUS System Usability Scale *(pp. 23, 67, 68, 70, 83)*

TLX Task Load Index *(pp. vi, 23, 67, 68, 70–72)*

UI User Interface *(pp. 49, 50)*

YAML YAML Ain't Markup Language or Yet Another Markup Language *(p. 6)*

1 Introduction

This chapter presents the work done in this dissertation, setting the context, purpose, and motivation of the study. It gives a context for the configurable code generation problems of Refinement of AADL Models for the Synthesis of Embedded Systems (RAMSES) and outlines the methodology and structure that guide the development of this thesis.

1.1 Context and Motivation

There is always a need for innovation, and consequently, technological progress is accompanied by an increase in system complexity [25], either software or hardware. This complexity is accompanied by enormous challenges in creating the solutions, particularly when software and hardware are adjacent to one another, such as is the situation when working with the robotics programming field.

For new users, robot programming can be especially daunting due to its extensive knowledge requirements and intricate integration of CPS [24]. Such systems, comprising computer-based programs, networks, sensors, and actuators, highlight the significant contribution of software development, which is responsible for the majority of the production cost of CPS [36]. Robotics, as a constituent of CPS, entails unique challenges in software and hardware integration, making problem identification late in the process very expensive [15].

Model-Driven (MD) approaches have proven to be effective solutions in this situation, offering the advantages of generation of high-quality code and results consistently [37]. The placement of the model at the center of the production process ensures that the developers are given a higher level of abstraction, while the complexity during the development of new systems is reduced.

1.1.1 AADL and RAMSES

Of all these MD techniques, the AADL is a strong modeling language well-suited to embedded systems [21]. It enables accurate description of hardware and software architecture

to support early validation and analysis of non-functional properties.

As shown by Borde et al. [14], the RAMSES project extends AADL to automatically generate source code for embedded systems. RAMSES, being a model-to-text transformation tool, enhances CPS software development quality and productivity by preventing human coding errors and accelerating the path from design to deployment.

> "With the ever increasing complexity of cyber-physical systems, RAMSES ensures trustworthy automation from design to deployment."

This project is a joint collaboration between Department of Computer Science (DI) NOVA, NOVALINCS, and Télécom Paris, unifying systems engineering know-how, formal methods, and embedded code generation.

1.1.2 Why Configurability is Necessary

While RAMSES utilization presents multiple advantages, there remains an issue concerning code generation: its decisions are hardcoded and rigid. Little is under control of the developer for elements such as code appearance, binding against specific APIs, or maintaining firm standards-based company guidelines. In industrial environments, where the need arises to reuse today's libraries and frameworks to uphold current standards, such a lack of flexibility presents a bottleneck, this inhibits not only initial integration, but also long-term system evolution and co-evolution, which are crucial for sustaining CPS over time [18].

Flexibility to customize generated code is imperative to encourage increased adoption by industry and to facilitate integration in diverse development environments [26]. With configurability, RAMSES can be set up to generate code that not only meets functional requirements but also conforms to organizational coding conventions and leverages accessible software assets.

1.2 Problem Statement

RAMSES does not have flexibility in its code generation process currently. Its generation strategies such as coding style conventions, library use, and API selection are fixedly embedded within its transformation rules. This lack of flexibility limits its application in industrial environments where projects rely on pre-existing company libraries and specific coding standards.

Currently, adapting RAMSES to different industrial contexts requires modifying its internal model-to-text transformation logic. This approach can increase maintenance effort and complicate integration with existing workflows, potentially limiting the broader adoption of the tool in diverse development environments.

1.3 Objectives and Contributions

The primary objective of this work is to enhance RAMSES configurability through the development and implementation of a configuration language. This language would externalize the parameters of code generation so that developers can tailor them to specific needs.

The key contributions of this thesis include:

- The definition of a DSL or configuration language for parameterizing the RAMSES code generation process.
- The introduction of mechanisms to enable customized C code generation, supporting various coding styles, library integrations, and API choices.
- The facilitation of reusing company libraries or APIs that already exist, enabling smoother integration of RAMSES into industrial development processes.

Contribution Summary:

- A configuration language for RAMSES
- Flexible and customizable C code generation
- Library reuse and integration support

1.4 Structure of the Thesis

This thesis is structured as follows:

- **Chapter 1:** Introduces the context, motivation, problem statement, and objectives of this research.
- **Chapter 2:** Provides a detailed overview of the state of the art, including Model-Driven Engineering (MDE), AADL, and existing code generation tools.
- **Chapter 3:** Identifies the challenges and requirements of creating a configurable code generator. Presents a prototype and evaluates feasability, implementation details and user feedback.
- **Chapter 4:** Describes the development plan for the configuration language.

2

Background and Related Work

In this chapter we will explore various tools related to the goal of this thesis: code generation configuration. While also explaining Acceleo, the main development tool used.

2.1 Model-Based Engineering (MBE) and AADL

Model-Based Engineering (MBE) has become a central methodology for the design of complex embedded systems . By putting high-level abstractions at the center, MBE enables engineers to manage system complexity through formal models rather than low-level code from the start [37]. This abstraction is particularly critical in embedded systems, where hardware constraints and timing requirements must be closely integrated with software behavior.

Several tool-supported methodologies, like NDT-Suite [23], show even more how MBE can be applied to real-world software engineering projects by offering methodological guidance and model-driven automation.

In the context of embedded systems, MBE facilitates early validation of design decisions, much earlier than hardware exists or code is written [22]. Engineers can model interactions, analyze performance bottlenecks, and verify compliance with safety and reliability standards, all at the model level.

One of the most important participants in this strategy is the AADL [21]. AADL is a formal hardware/software co-design modeling language. It gives precise semantics to model the architecture and behavior of embedded systems, ranging from processor bindings and memory layouts to communication buses and task scheduling.

AADL not only provides expressive architectural modeling constructs but also supports early analysis of non-functional properties such as timing, reliability, and safety constraints. Its formal semantics and tool support enable systematic verification of these properties during the design phase [21]. This is very well suited to industries such as aerospace, automotive, and defense, where such considerations are a given.

With AADL adoption, developers are able to early validate system architectures, preventing downstream integration risks and costly late-stage design modifications.

In this thesis, AADL is utilized as the base modeling language. Its formality and tool support, particularly within RAMSES, will facilitate automatic translation of abstract designs to the execution code and bridging of the system design and implementation gap.

2.2 RAMSES: A Code Generator for AADL

RAMSES is an M2T transformation tool with the ability to generate code from AADL models. Part of the greater Eclipse ecosystem, RAMSES automates the transformation of architectural models into deployable source code, effectively achieving the MBE dream of model-driven automation[3].

RAMSES now supports code generation in both **C** and **C++**. This makes it possible to use it in a broad variety of embedded development settings, depending on whether the target environment needs low-level procedural programming or more structured, object-oriented design paradigms.

The tool does this by systematically correlating AADL model elements to their corresponding code structures. Processors, threads, communication channels, and data components declared in AADL are mapped to their code counterparts, so much of the boilerplate and scaffolding code otherwise written by hand being done automatically.

Automation through RAMSES accelerates development and reduces human error, especially in large-scale embedded projects.

Even with these advantages, improvements can still be made. RAMSES' transformation logic is currently hardcoded, so developers have little control over customizing or fine-tuning the code structure generated without having to alter the tool itself. This rigidity becomes a performance bottleneck in projects that involve customized code structures, strict following of certain coding guidelines, or multi-variant code generation.

2.3 Code Generators in AADL and Beyond

While RAMSES plays a central role in the AADL toolset, it is by no means alone in the world of model-based code generation. There are long-established solutions both inside and outside the AADL universe with their own capabilities and niches.

2.3.1 Simulink Code Generation For Embedded Systems

Simulink is a flagship Model-Based Design solution, particularly in control systems engineering, developed by MathWorks [4]. In comparison with the tightly integrated

AADL inherent RAMSES, Simulink is backed by a graphic modeling framework of dynamic systems, and the production of code becomes straightforward with software like Simulink Coder and Embedded Coder.

Key aspects of Simulink code generation are:

- **Model-Based Design:** Control systems can be graphically designed, simulated, and validated by engineers before code generation.
- **Template-Based Generation:** Code is generated from pre-defined templates to enable integration into existing software platforms.
- **Customization and Extensions:** Developers can customize generation patterns and integrate generated code into larger legacy codebases.

Simulink is especially well-suited for rapid prototyping and tight integration with hardware-in-the-loop testing, and thus it is a favorite among automotive and aerospace industries [5].

2.3.2 OpenModelica: Modelica-Based Code Generation for System Simulation

OpenModelica is an open-source Modelica language-based modeling, simulation, and code generation software used intensively for system and physical modeling [33]. It generates simulation binaries and C code that precisely represent Modelica models and support complex system dynamics and numerical analysis [34].

Configuration options are available through Modelica annotations and compiler flags, allowing control over simulation parameters and some aspects of code generation. These are, however, mostly simulation-related settings and not related to control of the level of source code organization, naming, and coding style.

Code generation in OpenModelica prioritizes the correctness and performance of the resulting simulation code and provides little support for adherence to a given coding standard or legacy code base [35]. The major facility of the tool is to create efficient executable simulation models rather than to be highly configurable with respect to code generation output.

2.3.3 OpenAPI Generator: Configurable Code Generation Beyond Embedded Systems

OpenAPI Generator is an open-source tool that generates client SDKs, server stubs, and documentation from OpenAPI specifications [32]. Supporting over 40 languages and frameworks [31], it is widely used across software projects.

Generation is controlled via configuration files (JSON or YAML Ain't Markup Language or Yet Another Markup Language (YAML)) that specify package naming, class prefixes, data type mappings, and code style, enabling consistent architectural and coding

standards [28]. The tool's template-based system uses customizable Mustache templates to define code output, allowing adaptation to legacy code, custom logging, or specific frameworks. Plugin mechanisms and hooks enable further customization during generation [30, 29].

This flexible, configurable approach contrasts with RAMSES's more rigid, hardcoded transformations.

2.3.4 RAMSES vs. Other Code Generators

To better understand how RAMSES holds up against the competition in terms of code configuration, Table 2.1 was created based on singular tool testing and interviews with developers.

Table 2.1: Code gen configuration feature comparison based on author's analysis of documentation and tool behavior

Feature	Simulink	OpenModelica	OpenAPI	RAMSES
Identifiers[1]	Yes	No	Yes	No
Legacy Code Integration	Yes	No	Yes[2]	No
Generational Hooks	Yes	Yes[2]	Yes	No
Traceability	Yes	No	Yes	Yes[2]
Reporting	Yes	No	Yes	Yes[2]
Protected Areas	Yes	No	Yes[3]	No
Inline Functions	Yes	Yes	Yes[4]	No
Dead Code Elimination	Yes	Yes	Yes[4]	No
Comments	Yes	No	Yes[2,4]	No
Compliance Support	Yes[2,3]	No[3]	No[3]	No[3]

As can be observed in Table 2.1, features like dynamic Identifier naming and legacy code integration are crucial, yet only present in some of the analyzed solutions, these features work well to achieve compliance standards imposed in the industry [16, 17], which is not natively supported by most apps like OpenModelica or RAMSES.

Generational hooks[1] are extremely useful since they allow for automated code customization before or after code generation, this explains why they are common in most code generators evaluated, though absent in RAMSES. Protected areas serve a similar function in the sense that they allow the user to customize a certain section of code only,

[1] Names of Functions, Classes, Variables, etc
[2] User-driven process (not entirely automatic)
[3] Normal code generation *might* generate compliant code, but its not very certain.
[4] Limited functionality or abstraction.
[1] Predefined extension points that allow developers to insert custom code or override generated behavior without modifying the generator itself.

not a complete script. Simply put, generational hooks modify artifacts before/after code is written and protected areas deny code modification in a specific area.

Traceability is a crucial feature that enforces coherence between model and code, essentially defining the guiding thread[2] of the project. This is very sought after in a modeling context and ties well with report generation and code comment management, unfortunately, these three features are implemented only at a basic level in the analyzed tools.

Code optimization is also an important area, the ability to remove unnecessary code or shorten it proves to be useful in varied contexts, most tools provide a way to automatically optimize the generated code. RAMSES lacks such capabilities.

Overall, Simulink, a commercial high-end software, surpasses its rivals in all key aspects of code generation configurability. Its support the most sought after configurations give it is an end-to-end solution widely used in applications needing both flexibility and performance, but not on AADL.

Conversely, OpenModelica is missing a number of key points of configurability, reflecting both its complementary focus and level of maturity for code generation functionality. OpenAPI Generator, although even providing a more user-driven process in some cases, it still misses on some key features. RAMSES, in turn, presently falls short on all features, with inflexible, hardcoded transformations that curtail its usability and controllability by users.

This comparison reveals, yet again, the motivation for this thesis: researching means by which RAMSES can be enhanced through the introduction of greater configurability and extensibility, and thereby narrowing the gap with more mature tools in the domain.

The features found in table 2.1 are taken, not just from the code generators observed, but also from the wants and needs of the industry. The following feature model, present in Figure 2.1 was the outcome of that research.

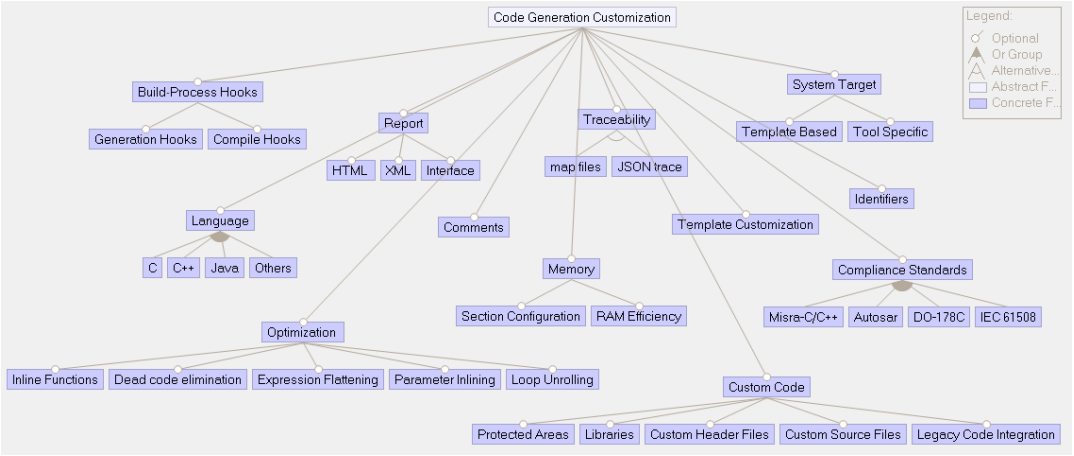

Figure 2.1: Code Gen Config Feature Model

[2]The coherent, central line of reasoning, logic, or narrative that connects all parts of a project or document.

With this, we can have a clearer look at the broader picture. The features suggested are not just the result of analyzing existing code generators like Simulink, OpenModelica, and OpenAPI Generator, they are also derived from a synthesis of industry demands and recurring pain points observed in real-world development environments.

Figure 2.1 organizes these configurability aspects into a feature model, grouping them into categories such as Optimization, Traceability, Compliance Standards, Memory Configuration, and Custom Code Integration. The model also highlights optional, alternative, and concrete features that modern code generation tools must support to be competitive and practical across diverse application domains, from automotive (e.g., AUTomotive Open System ARchitecture (AUTOSAR), Motor Industry Software Reliability Association (MISRA)) to safety-critical systems (e.g., DO-178C, International Electrotechnical Commission Standard (IEC) 61508).

Additionally, this model also possesses some constraints when dealing with certain features, namely:

- MISRA-C/C++ implies C or C++
- AUTOSAR implies C
- DO-178C implies C or C++
- IEC 61508 implies C or C++ or Java
- Template Customization implies Template Based

This essentially means that most code standards are locked to one or more programming languages, if those programming languages are not selected, the standards does not apply. Similarly, Template Customazation can only be applied to Template Based solutions.

2.4 Acceleo and Model-to-Text Transformations

To counter the configurability limitations observed in tools like RAMSES, we turn to specialized M2T transformation technologies. Among these, Acceleo is highly promising.

2.4.1 Acceleo: An Overview

Acceleo is an open-source, template-based Eclipse family M2T transformation tool. Its thought model is based on the mapping of formal models (typically in Eclipse Modeling Framework (EMF)) to text artifacts like source code, documentation, or configuration files [6].

Major benefits of Acceleo are:

- **Template-Based Transformation:** Specified templates describe how the elements of a model should be translated into textual form.

- **Strong Eclipse Integration:** Acceleo offers robust integration with the Eclipse IDE, providing instant feedback, syntax coloring, and incremental generation.
- **Structured Code Generation:** Well suited for generating structured, maintainable C/C++ code from high-level models.
- **Project bias:** RAMSES currently uses Acceleo as a code generator so its logical to improve the work done as opposed to the integration of a new technology.

Acceleo gives developers the ability to tweak code generation patterns, making the generated codebase more flexible and maintainable.

2.4.1.1 Acceleo example

Acceleo code is generated from .mtl templates, where it is defined how model elements are translated into text. Suppose the model contains a Class with the name MotorController. A basic Acceleo template might look like:

```
[template public generateClass(c : Class)]
public class [c.name/]
{
 // class body
}
[/template]
```

This would generate:

```
public class MotorController
{
 // class body
}
```

This example illustrates how Acceleo uses MD templates to produce structured and consistent source code from high level models.

2.4.2 Acceleo's Role in This Thesis

For this project, Acceleo serves as the basis for enhancing RAMSES' configurability. Through delegating transformation logic to Acceleo templates, we have the aim of:

- Isolate transformation rules from RAMSES' internal code.
- Allow easy extension and modification of code generation patterns.
- Facilitate adherence to industrial standards such as MISRA C/C++.

This plan promises to transform RAMSES into a more flexible and maintainable toolchain component from one that is rigid code generating.

2.5 Existing Work on Configurable Code Generation

The search for flexible and customizable code generation is not unique to this thesis. In most domains, tools and techniques have been created to solve the problem of generating high-quality, customizable code from models.

2.5.1 Template-Based Approaches

Template-based code generation remains the foundation in this field. Some good examples of such tools are **Acceleo** and **Simulink templates**:

- **Acceleo** allows explicit control of the structure and style of the generated code, making it highly suitable for projects in which compliance with some coding standards or architecture patterns is essential [6].
- **Simulink Templates** offers programmers the means to declare patterns of reusable code, with uniform look and feel across several projects and support for custom toolchains and legacy systems [7].

These approaches allow programmers to shape the auto-generated code towards project-specific applications without downgrading underlying models bridging the gap between automated generation and hand-coding, combining efficiency with flexibility.

2.5.2 Hook Functions in TargetLink

TargetLink, another market leader in code generation tools, comes with the concept of **hook functions**: pre-compiled points of extension within the generated code that allow developers to plug in their own logic [8]. The facility is most handy in a number of situations. For example, it eases the integration with legacy APIs or platform-dependent libraries and allows developers to add extensions without altering the primary generated code.

In addition, hook functions have the benefit of being customizable without compromising maintainability or upgradability of the generated code. When models evolve, code under it can remain unchanged while introducing custom logic using these extension points. This solution offers a clean trade-off between extending the generated code and offering its long-term maintainability with less effort for future upgrades.

2.5.3 OpenModelica and Multi-Variant Generation

OpenModelica introduces a higher degree of configurable generation with its support for **multi-variant code generation**. Through this, engineers are able to:

- Create multiple variants of code based on a common base model.
- Tailor outputs for various deployment contexts, hardware configurations, or performance constraints.

This variability is completely indispensable in automobile or aircraft production companies, for example, where a single product line might encompass several hardware targets or safety classes.

2.5.4 The Case for Configurability in RAMSES

Despite its strengths, RAMSES currently has no mechanism for fine-grained extension and configuration. Specifically:

- Transformation rules are hard-coded, which restricts flexibility.
- There is no native support for multi-variant generation or integration points like hook functions.

Including configurability in RAMSES would offer several benefits. It would facilitate the generation of custom code for different deployment environments, making it easier to adapt to specific hardware environments or performance requirements. In addition, the flexibility would simplify maintenance and development of the transformation logic, allowing the tool to better support changing development needs. Finally, by making RAMSES more configurable, it would be easier to interface with industry standards and legacy systems, rendering the tool flexible and applicable in high-speed industries.

By adopting template-based generation, RAMSES can evolve into a dynamic, future-proof tool to meet growing embedded system development demands.

2.5.5 Towards MISRA C/C++ Compliance

A central element of code generation in configurable code generation, particularly in the field of safety-critical application domains, is to generate **standard-compliant code**. Strict requirements for safe, portable, and reliable embedded software are presented by the MISRA C [16] and C++ [17] standards.

Compliance to MISRA plays several principal roles: it enhances software safety by minimizing the likelihood of undefined behavior and runtime errors, guarantees that development processes meet the high standards demanded by industries such as the automobile and aerospace industries where in some instances compliance is mandatory, and is readily compatible with existing toolchains, as most static analysis tools are tailored to enforce MISRA rules.

As we integrate configurable generation facilities into RAMSES, we shall ensure that code generated is MISRA C/C++ compliant.

Flexible code generators need to not just conform to project requirements but also apply vital industry standards such as MISRA to guarantee safety and reliability.

While current solutions to code generation customization exist, they don't apply to AADL directly and the most powerful out of all the solutions studied, Simulink, is closed source, which does not solve our inherent problem: build a configuration language for RAMSES. This would make the configuration language the only open-source codegen configurability layer for AADL.

3

Challenges and Requirements for a Configurable Code Generator

This chapter explains why configurability is no longer optional, and how traditional code generation pipelines must evolve. In addition, a prototype of the configuration is also presented along with a system evaluation plan and research limitations.

3.1 The Inflexibility of RAMSES: A Barrier to Industrial Integration

RAMSES has served as a robust model-to-code generator for AADL based systems, yet it suffers from a fundamental architectural constraint: it assumes a uniform target environment. This assumption does not hold in real-world industrial projects, where system heterogeneity, legacy integration, and domain-specific standards define a constantly shifting context.

The core issue lies in RAMSES' transformation pipeline: it entangles policy decisions (naming, structure, integration style) with generation logic. These decisions, hard coded in ATL transformations, reflect the assumptions of RAMSES' authors more than the needs of end users [9]. Altering them involves modifying the transformation source itself, often a hard and error-prone task [27].

Consider a simple use case: a company mandates that all task-level functions use 'snake_case' and include a 'COMPONENT_' prefix. RAMSES, which might generate 'ComputeTask' by default, offers no way to enforce those rules. A change in naming becomes a traversal through ATL templates and helpers. This is not scalable, and in safety-critical software, it is not acceptable.

Furthermore, beyond naming, decisions about memory allocation models, system initialization flows, and error handling behaviors are equally rigid. There is no declarative layer that allows users to steer generation outcomes according to organizational needs or evolving constraints.

3.2 Industrial Realities and Pressures

To understand why this rigidity is problematic, we must shift perspective from the generator to the organization that consumes it. In industry, generated code is not ephemeral: it is versioned, peer-reviewed, statically analyzed, tested, and in some cases certified. It coexists with handwritten code, interfaces with platform-specific services, and must evolve alongside requirements.

3.2.1 Compliance, Traceability, and Certification

Generated code must often comply with domain-specific standards such as:

- **MISRA C/C++:** Imposes constraints on memory usage, naming, control flow, and portability [16] [17].
- **DO-178C:** Requires traceability, tool qualification, and clear derivation from high-level requirements [2].
- **International Organization for Standardization (ISO) 26262:** Enforces safety-related development practices and documentation [19].

In these environments, code generation must do more than "just work". It must be explainable, auditable, and deterministic. Developers must be able to trace a generated function back to a model element and forward to a specific runtime behavior.

3.2.2 Integration with Legacy Codebases

Most industrial systems are not built from scratch. Code generators must work alongside:

- Legacy libraries with non-negotiable APIs that can potentially evolve overtime, as they are third party.
- Hardware abstraction layers that impose structural patterns.
- Existing software architecture rules (how modules communicate, how tasks are organized).

A code generator that cannot adapt to these constraints is often sidelined in favor of manual glue code or post-processing scripts. These scripts, in turn, introduce maintainability challenges and break traceability chains.

3.2.3 Developer Ergonomics and Maintenance

Even the most advanced generator will eventually produce code that is read (and possibly modified) by a human developer.

If developers can't read or rely on the generated code, they will stop trusting it altogether.

Poor formatting, ambiguous naming, or surprising control flow all reduce the utility of generated artifacts, leading teams to "lock" generated files and prohibit modifications: an anti-pattern that defeats the promise of model-driven engineering.

3.3 Why Configuration Matters

To resolve the issues above, we must introduce a new abstraction layer: one that separates **what** is generated from **how** it is generated. This is the role of a configuration language.

A configuration language provides a structured way to express **generation policy**: the set of rules, conventions, and constraints that tailor code to its industrial context. Importantly, it allows these policies to be:

- **Externalized** from the transformation logic.
- **Composable** and layered across project variants.
- **Validated** for correctness before code generation begins.

Such language enables a fundamental shift: from a monolithic, one-size-fits-all generator, to a configurable and extensible platform that adapts to its environment.

3.4 Characteristics of an Effective Configuration Layer

Designing this kind of language is challenging. It needs to balance expressiveness, ease of use, and seamless integration. Based on industrial feedback and analysis of RAMSES issues, the following characteristics are proposed:

- **Declarative, Not Imperative**: Users should describe **what** they want ("all functions must use snake_case") rather than **how** to achieve it. This aligns with the model-driven philosophy and supports better configuration analysis.
- **Human-Readable and Tool-Accessible**: The configuration format (proposed DSL) should work well with version control, support difference review, and remain readable to engineers. It must also be machine-readable for validation and generation.
- **Functionally Equal to the Default Code**: The newly generated code should remain functionally the same as without the configurator. Meaning that it should produce the same practical result with or without the configuration, excluding performance metrics.

- **Configures the Generator, Not the Model**: Configuration keys should influence how the code is generated, without requiring changes to the input models. The DSL operates alongside the model, guiding the generators behavior (naming, structure or implementation strategy) based on domain concepts such as `thread`, `port`, or `data component`. Configuration keys should remain semantically aligned with modeling elements but their primary role is to modify generator logic, not to modify or extend the models themselves.

- **Validated and Error-Tolerant**: Invalid configurations should produce clear diagnostics before generation starts. Where possible, defaults and fallbacks should be available to prevent blocking workflows.

- **Extensible and Portable**: The configuration will be easily extensible thanks to its Ecore background, allowing for easy inclusion of new features. At the same time, since the language model is detached[1] from RAMSES, it can be easily included in other projects, allowing for a quicker development of a configuration language once this one is built.

These characteristics will make the configuration language robust, maintainable, and suitable for industrial use, allowing for flexible customization of code generation without compromising model integrity or introducing unnecessary complexity.

3.5 Beyond Code Formatting: What Configuration Should Control

While naming and formatting are important, a powerful configuration system must go further. The following dimensions should be within scope:

1. **Artifact Naming and Structuring:** Control over file names, folder layout, and Identifier styles.

2. **Component Mapping Rules:** Declarative rules that assign AADL components to target platform concepts (RTOS tasks, processes, etc).

3. **Code Instrumentation:** Hooks for logging, tracing, or runtime checks (insertion of ‘assert‘ or instrumentation macros).

4. **Conditional Feature Flags:** Ability to enable or disable parts of the generator (generate test stubs, insert wrappers).

5. **Code Documentation:** Generation of reports or toggle of code comments.

[1]The language model (Ecore) itself is not locked exclusively to RAMSES and can be used in other projects, however, that implies having to code the logic for each feature according to project specifics.

Configuration is More Than Style

While formatting is the most visible aspect of configurability, its true value lies in controlling semantic properties of the generated code: platform binding, integration, traceability, and lifecycle.

3.6 Proof of concept

In order to better understand the whole concept of the DSL to be built during this thesis, a prototype that encompasses a specific feature needed in the RAMSES tool was built with the intent of showcasing the usage of the configuration language, in a controlled environment.

3.6.1 How it works

The selected feature to be implemented by the prototype was the Identifier Modification option, which essentially means having a higher control over the generated function, variable, and class names in the generated code. This features was chosen since it is fairly straight forward to implement and also decently portable, as discussed in section 3.4.

Lets take the following example:

```
class MyNode : public rclcpp::Node {
 public:
 MyNode() : Node("my_node") {
  (...)
 }
```

A simple Node class has the class Identifier `MyNode`. In the case of RAMSES, MyNode would be inherited from the AADL model, meaning that, in order to change the code nomenclature of MyNode, we would have to either manually change the generated code or changing the name of the node in it's corresponding AADL model file and then regenerate the code.

Both options are feasible in this case, however, with growing system complexity this small change becomes incrementally harder and more prone to errors.

The developed prototype focuses directly on that front, providing an adjacent configuration model that works alongside the main one with the goal of delivering that missing configuration option, here's how it works:

1. The placeholder code is built, in this case it was from a taskset style project[2]

2. The target naming convention is selected in the configuration model

[2]This project is a Domain Specific Modeling Language (DSML) that communicates task sets and automatically generates C code using Acceleo, targeting the RT-POSIX real-time glsAPI.

3. The final code is generated with the now correct Identifier nomenclature.

For a clearer understanding of the flow of the prototype, lets take a look at figure 3.1:

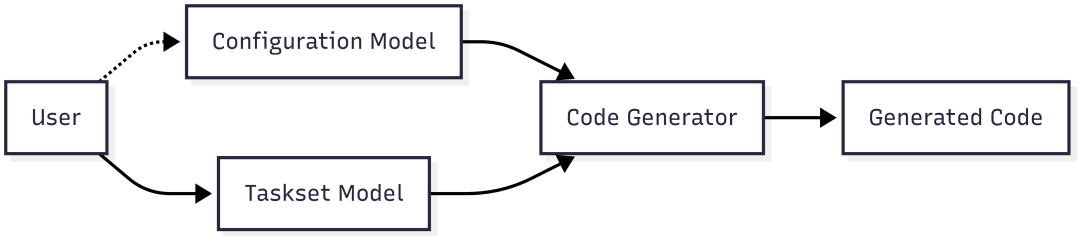

Figure 3.1: Flowchart of the prototype

The user interacts with the main model (Taskset in this case) and *can* interact with the configuration model as well, if that's the case, the user will see changes in the code in accordance to the options selected in the configuration model.

The configuration model is not tightly coupled to the main workflow and is not mandatory, it works alongside the main model but provides extra options and choices to the user in terms of code customization, effectively elevating a bit more the code quality of the given taskset generated code.

3.6.2 Practical Example

Let's take a look at a practical example of the prototype. Since the placeholder code has already been defined and altered to allow for dynamic Identifier changes, all we have to do is select the naming convention on the configuration model.

Figure 3.2: Selection of the naming convention

As we can observe from Figure 3.2, the prototype presents a rough and unrefined way of user interaction, in order to change the configuration the user must change the configuration model itself via its properties, which is not the end goal for the final product but works well in testing/controlled environments.

The current selected option in Figure 3.2 is the default option, which will not modify the file (see Figure 3.3)

Taking in the result from Figure 3.3 we can get an idea of the default, unaltered code generation. However, our goal is not simply to generate code. We can select a different naming option via the configurator model, save, and regenerate the code. The change is clear.

```
23 void T1();
24
25 // Initialization of data for task T1 (periodic, period = 1000 ms)
26 thread_config_t T1Info = { 1000, 0};
```

Figure 3.3: Result of the code generation in the default form

```
23 void t_1();
24
25 // Initialization of data for task T1 (periodic, period = 1000 ms)
26 thread_config_t t_1_info = { 1000, 0};
```

Figure 3.4: Result of the code generation with the snake_case naming option

Comparing Figure 3.3 with Figure 3.4, we can clearly see where the configuration model applies its modifications: always on the Identifiers and never on anything else, especially not on the keywords[3].

The idenfitiers (in this case the name of the function[4] and the name of the variable[5]) change seamlessly according to the selected naming convention in the configurator. This is the expected behavior for the features to be implemented, they should provide additional value while also keeping the base generated code available if the user does not wish to modify it at all.

3.6.2.1 User Feedback

In order to test the functionality and receive a third party opinion, a simple user testing was done with two users, specifically RAMSES developers that work directly with AADL. The procedure was simple and direct, users were instructed to the premise of the prototype, then they would perform tasks (in this case just one task: change the Identifier name to X) and then they would be questioned in the usefulness of the feature, the applications it could have and overall thoughts.

Upon doing this testing, the users defined the feature as a flexible addition to coding generation with many application to real life and industry scenarios. Users also noted that this functionality improves information treatment between teams as different people have different ways to write models names, the prototype proves that naming normalization can also improve team cohesion.

Users commented that this functionality supports the ability to create multiple versions of functionally equivalent code just with naming differences. This also aligns with the required feature of supporting coding standards for the selected language[6], given that

[3]Keywords like *Void* are common and easy to identify. In this observed case however, in both Figures 3.3 and 3.4 we see that *thread_config_t* is also present but this is a *Struct*, which, while not an exclusive keyword, it serves a purpose like *Void* or *int* and is **not** an Identifier.

[4]T1

[5]t_1_info

[6]Currently C++, however there are plans to include C.

some languages require Identifiers to use a certain nomenclature [10, 11], which can clash with AADL node names.

Overall the prototype, although simple, was well received and proved to give value for the AADL to code pipeline.

3.6.3 Technical Details

Even though its application is specific, the code developed for the prototype is fairly generic. This means that most of the logic and code used for this prototype can be ported directly to RAMSES without much issue, effectively implementing the feature, albeit testing will still be performed.

The naming convention change logic is pretty straight forward considering Identifiers, and those come from model properties. From there the properties that represent Identifiers are formatted as they are being added to the code, effectively making sure the main model stays untouched while the properties extracted are modified into the desired outcome. This makes sure only the specific selected properties are modified and the rest of the code generates as normal.

The algorithm used for the naming transformation has many fallbacks that make it so even if a property name is especially complex, cases like short names are handled with care and specific styles like snake_case, while having complex patterns, can be easily defined with Regex. [39] As the configuration model is built on Ecore, the modification of naming styles is not possible directly by the user. However, if instead we wanted to **add** a new naming style, that is very much possible, however it required a quick modification of the Configurator's Ecore model and the Acceleo naming format algorithm, as in Figure 3.5

Figure 3.5: Flowchart of adding a new naming convention to the configurator

Figure 3.5 explains the steps in a clear but still abstract manner, the Ecore model needs to be modified to include the new naming style, which in turn will generate the coding logic that we can use in Acceleo to transform the Identifiers by improving the current name formatting algorithm to accommodate the new change.

3.6.3.1 Multiple Models in Acceleo

Working with 2 Ecore models was not very straightforward. Although Acceleo provides the resources needed to work with multiple Ecore models, Eclipse itself does not allow for multiple input models in a single run configuration [12]. To work around this, a custom *main()* method was made to register the metamodels into the resources, effectively allowing for the usage of both metamodels. This issue, while small, presented a fairly

decent deal of complications for the development of the prototype, specifically because Acceleo documentation on the matter was scarce and community forums were unhelpful.

Nevertheless, this solution allows not just for the usage of multiple models in this prototype but also in any other projects that need an Acceleo program with 2 or model Ecore models as inputs.

3.7 System Evaluation

To assure that the DSL meets expectations [38], the system will be evaluated in various forms, mainly:

- The DSL exists and produces results.
- Before and after comparison (with code configurator vs without).
- Usability evaluation.
- Configuration variants evaluation.
- The newly generated code remains functionally the same as before[7].
- Various testing procedures during and after development.

Ideally the end product should meet these requirements to be considered a success. Each one of the requirements ensures a different, core quality of the configuration system, such as correctness, usability, or robustness.

3.7.1 DSL existence

This requirement will be fairly straightforward to prove since it implies a direct usage of the DSL to produce results. This can be proven alongside the other requirements as they depend on it. However, a system wide testing will quickly prove the existence of the configuration language and its impacts, specifically with the testing of all features developed and their results.

3.7.2 Before and After Comparison

The core idea of this configuration language is to modify the output code of the code generator according to user needs. The best way to prove that this goal was achieved is to compare how users would perform the same task before the configuration language was implemented and after. This direct comparison will evaluate the direct impact of the configuration language on development and will ensure that the developed solution benefits the system.

[7] Meaning that it performs the same practical task as before the DSL implementation.

To strengthen the evaluation, a small controlled experiment involving developers familiar with model-driven engineering and code generation tools (EMF, Acceleo) will be performed. Participants will be asked to complete predefined tasks under both conditions.

3.7.3 Usability

The configuration language should be easily usable, even by end users with limited programming experience, to ensure this requirement is met, clear documentation, consistent syntax, and terminology aligned with the application domain should be prioritized. This requirement will be confirmed by comparing the interaction with the configuration language of users with varying levels of technical expertise within the RAMSES ecosystem, including RAMSES developers, testers, and other stakeholders familiar with AADL, allowing the comparison of usability and learning curve.

3.7.3.1 User Evaluation Methodology

Participants will be given predefined configuration tasks (changing naming schemes, visualizing traceability, etc) and asked to perform them with minimal guidance.

The evaluation will collect both quantitative and qualitative data:

- Task completion time
- Number and type of errors
- Subjective feedback on ease of use
- Observed learning curve
- Type and gravity of misunderstandings

To further support analysis, two standardized questionnaires will be used:

- The System Usability Scale (SUS) to measure perceived usability.
- The NASA TLX to evaluate the perceived workload and cognitive effort involved in using the DSL.

This setup will help assess whether the DSL meets its usability and accessibility goals across a broad user spectrum.

3.7.4 Configuration Variants

To assess the flexibility and robustness of the configuration language, distinct configuration files will be tested. These scenarios will vary in complexity and structure to evaluate whether the generator can handle diverse inputs while maintaining correctness and coherence in the generated code, effectively testing the limits of the configuration language.

3.7.5 Functionally Equal

To make sure the code generated still retains the same practical functionality, a few things need to be taken into account, firstly, the code generator needs to be modified just enough to produce the desired results of structure, naming, etc while still keeping the same functionality as before the implementation, essentially keeping the semantics intact. Monthly testings, including Regression and Integration testing, will also ensure that the code remains functionally the same by running selected examples and doing direct code comparison, however, comparisons will focus on runtime behavior rather than raw text equivalence. This will give us the confidence that the new code behaves the same as the old one.

The baseline for comparison will be the unmodified RAMSES generator. This allows for a clear assessment of what configuration capabilities are introduced in terms of productivity, flexibility, and clarity.

3.7.6 Testing Procedures

Tests will be conducted during and after development to ensure the configuration language behaves as expected and that the practical usage of the generated code remains the same. A variety of tests were chosen, specifically:

- Unit Testing which will be performed by the developer. These tests focus on individual transformation rules or components of the generator to make sure developed features work as intended.

- Regression Testing are also handled by the developer as they are used to ensure that new changes do not break existing functionality. They target both previously implemented configuration options and code generation rules.

- Integration Testing will also be performed by the developer and checks whether different features of the generator work together correctly. It focuses on feature interactions and overall system behavior, especially when configuration options are combined.

- Validation Testing is conducted involving actual users and project stakeholders. Its purpose is to validate that the features implemented match user expectations and project goals. Participants will evaluate whether the generated code behaves as expected and whether the configuration language is understandable and practical for its intended use.

Each of these testing strategies plays a specific role in evaluating the system, they ensure that the features work and users understand them. The outcomes of these tests is extremely important for the final evaluation of the prototype.

3.8 Research Limitations

Although the configuration language includes many features and has been designed with flexibility, usability, and maintainability in mind, this work has some limitations.

- **Limited Scope of Configurable Aspects:** The proposed DSL focuses primarily on naming conventions, traceability, and coding standards. More advanced features such as behavioral configuration, cross-component constraints, or real-time performance tuning fall outside the scope of this work [1]. Similarly, certain features might be prioritized or excluded based on client feedback.
- **Restricted Evaluation Sample:** Usability and functional testing rely on a small set of representative examples and a limited number of user interactions [13]. While including users with varying levels of technical expertise is being kept in mind, the results may not generalize across all industrial contexts or teams.
- **No Legacy Tool for Direct Comparison:** Since no previous configuration language exists for RAMSES, or AADL for that matter, before and after comparisons will rely on manual, potentially subjective assessments of workflow complexity. This limits the ability to measure productivity gains with high precision.
- **Integration Constraints:** The configuration system is designed to integrate into the existing RAMSES toolchain, which constraints how the generator behavior can be modified. Deep generator reworks or architectural transformations are intentionally excluded to maintain compatibility.
- **Third Party Library Issues:** The inclusion of third party code that is not affected by the configuration language might break compliance standards since the transformations only affect RAMSES-generated code. This limitation is hard to overcome since it depends solely on those third party libraries introducing standard compliant code.
- **Scalability Not Fully Tested:** The DSL should performs well on example models. Its performance and maintainability might not be validated on extremely large or complex input systems, which are typical in some industrial environments. This might not be a constraint if such models are later used for testing.

These limitations do not compromise the core contributions of this work, but they highlight areas for future development, such as broader configuration domains, wider user studies, and deeper toolchain integration.

3.9 Toward a Generator for System Integrators

RAMSES has proven to be a robust and effective tool for model-to-code generation, particularly within academic contexts and controlled environments. However, as embedded

systems development increasingly intersects with regulatory and industrial demands, the expectations placed on code generators have grown. Beyond correctness, they must now offer adaptability, traceability, and long-term maintainability to meet evolving project requirements.

Introducing a configuration language is not merely a feature, it's an architectural shift. It allows users to define their generation context without touching generator internals, reducing risk, increasing reuse, and enabling automation across product lines.

4

Implementation and Integration into RAMSES

This chapter describes the technical implementation of the proposed configuration language for RAMSES. It details how the insights from industrial collaboration informed the architectural decisions and how each feature was implemented.

4.1 Industry Partner Meeting

The first meeting with an industry partner revealed to be very useful with many insights to note, particularly that a few features were highlighted and prioritized in order to maintain a good development speed and an overall better plan. Most of the previously thought of features (Identifiers, Traceability, Report, etc) were endorsed, however, a new feature was suggested by an expert Robot Operating System (ROS) developer: the separation of ROS specific code and C++ specific code, meaning that code that is specific to ROS can be isolated as it is very similar along the various generated files. This is essentially done with an interface that implements ROS code on C++ only files, which effectively serves as a separation of concerns.

Beyond that, the meeting also served as a sneak peak to what the final RAMSES product would look like, including every gymmic, option, page and also a code generation dedicated section that ties perfectly with the current work.

This meeting provided extremely valuable insight as the industry partner has been using ROS for over seven years, effectively distilling their experience into actionable knowledge.

4.2 Implementation Flow

In order to better understand what features were implemented, when and why, the implementation was planned and executed by layers: each layer depends on things done

on previous layers, hence some features were developed before others.

Table 4.1: Feature dependency table

Feature	Depends On	Notes
Identifiers	-	Core to code structure
Comments	Identifiers	Low implementation cost early
Traceability	Identifiers, Comments	Hard to retrofit later
Report	Traceability	Uses trace data
Standard Compliance	Report, Traceability	Enforce compliance early
Code Quality Analysis	Report	Needs stable generation logic
Business Logic Separation	-	Enables ROS decoupling
File Management	All	Facilitates file management operations
Generation Hooks	-	Adds autonomy to generation
Multi Configuration	-	Easier configuration management
Build Integration	-	File generation management
Legacy Code Integration	Business Logic	Most architecture-dependent
Target Operating System	Business Logic	Increases flexibility

Table 4.1 presents the planned order of feature implementation based on their dependencies. Some features required elements developed in previous steps for instance the Report which uses Traceability features. This dependency structure prevented rework by implementing features that provide necessary infrastructure before higher-level functionalities.

4.2.1 Core and Metadata Layer

The Core and Metadata Layer (Table 4.2) includes foundational features that establish the basic code structure and metadata necessary for subsequent processing. Features such as Identifiers and comments form the backbone of the generator, while traceability and reporting provide essential tools for debugging and verification. Their relatively low to

medium complexity allowed them to be developed early, enabling smooth integration of more advanced features later.

Table 4.2: Core and Metadata features and their complexity

Feature	Complexity	Notes
Identifiers	Low	Nearly complete.
Comments	Very Low	Can parallel Identifier completion.
Traceability	Medium	Requires structured tagging throughout the generator.
Report	Low-Medium	Build once traceability is present; not deeply complex.

This first layer served as a base for other features since it has fairly straightforward but very useful features.

4.2.2 Structural Optimization Layer

The Structural Optimization Layer (Table 4.3) focuses on improving code quality and efficiency. Standard compliance ensures generated code meets coding norms and best practices. Dead code detection and memory optimization reduce code bloat and resource consumption but require stable metadata and analysis frameworks built in previous layers. These features have higher complexity, reflecting the need for careful analysis and manipulation of generated code.

Table 4.3: Structural Optimization Layer

Feature	Depends On	Notes
Standard Compliance	Report, Traceability	Defining and checking rules requires steady effort.
Dead Code Detection	Traceability, Report	Done by Cppcheck.
Memory Optimization	Dead Code Det, Traceability	Impacts core data structure generation. Error-prone.

The structural layer cements what was done before and adds to it with an optimization focused approach.

4.2.3 Architecture Layer

The Architecture Layer (Table 4.4) introduces features that abstract and modularize the generated code to support scalability and maintainability. The node interface abstraction separates platform-specific code (ROS dependencies) from core logic, facilitating reuse and easier updates. File management is a quality of life feature which is very sensitive as file naming and location must be consistent at all times. Legacy code integration is the most complex and architecture-dependent feature, involving interfacing with existing external codebases, which requires careful design and planning to avoid integration problems.

Table 4.4: Architecture Layer

Feature	Depends On	Notes
Business Logic Separation	Memory Optimization	Architecture-dependent. Needs careful planning.
File Management	All	Affects every file directly
Legacy Code Integration	Business Logic	Undefined scope and likely the most architecture-sensitive

The last layer of implementation has a higher degree of abstraction and requires a much more project architecture understanding than the previous layers, which is why it is done last and after every other feature layer is implemented. This allowed for the full focus to be on these core features that will have a much higher individual impact on code generation than the previous ones.

4.3 Implementation

As the implementation methodology was done iteratively and following the plan discussed in Section 4.2, the first feature to implement was the naming conventions.

> Implementation = "how RAMSES interprets and applies the DSL to affect code generation".

4.3.1 Naming Conventions

The implementation of naming conventions was straight forward since the demo developed in Section 3.6 was substantial part of the complete mechanism. Using Acceleo we are able to modify the properties coming from the model as they are added into the C++ file, in this case, the name is formatted according to the selected convention. The main addition to the prototype work was the introduction of dynamic naming conventions that depend on the selected language, for example: in C++, classes are declared in PascalCase, whereas

variables are declared in snake_case, the following generated code illustrates that very well:

```
 class ImageDisplay : public rclcpp::Node
{
 public:

 ImageDisplay(): Node("ImageDisplay")
 {

  image_subscriber_sub_ = this->create_subscription(...);

  nd_spinner_sub_ = this->create_subscription(...)
```

As shown in the generated code, ImageDisplay and image_subscriber_sub_ are both Identifiers but each one has a different format. In contrast, the previous implementation without configuration was strictly be *Image_Display* and *Image_Subscriber_Sub_*. These names come directly from the AADL model as ***Image_Display*** **and** ***Image_Subscriber***.

To summarize, the whole naming convention implementation consisted in:

- The ability to change Identifier (class, variable, method) formats dynamically from a list of implemented options (camelCase, UPPERCASE, lowercase, snake_case and UPPER_SNAKE_CASE).
- Based on the selected language, assign different styles to different Identifiers.
- Option to input a prefix or suffix in every Identifier[1].

This implementation was very important since the remaining features used it to the fullest, hence why it needed to be the first, on top of that, it improves readability and enforces language-specific best practices, which are critical in mixed-language projects (in the case of RAMSES: C++ vs ROS).

4.3.2 Comments

The implementation of dynamic code comments allow the user to get much more context on the code while preserving the ability to view raw, non-commented code. The way it works is pretty straight forward, the user can toggle the comments on or off, comments are dynamic according to their component and only provide essential info.

Figure 4.1 illustrates how the comments vary depending on the component, this also provides a degree of separation that helps developers understand where parts of the input model are being implemented.

[1]This affix is not affected by the naming convention.

```
// Constructor: Initializes all components
ColorTracker(): Node("ColorTracker")
{
    // ---------------- CtSubscriber Component ----------------
    ct_subscriber_sub_ = this->create_subscription<std_msgs::msg::Int32>("/
        10, std::bind(&ColorTracker::ct_subscriber_callback, this, std::pla

    // ---------------- CtPublisher Component ----------------
    ct_publisher_pub_ = this->create_publisher<std_msgs::msg::Int32>("/colo
    ct_publisher_timer_ = this->create_wall_timer(500ms, std::bind(&ColorTr

    // ---------------- Image_broadcaster Component ----------------
    image_broadcaster_pub_ = this->create_publisher<std_msgs::msg::Int32>("
    image_broadcaster_timer_ = this->create_wall_timer(500ms, std::bind(&Co
```

Figure 4.1: Example of generated code with comments

More examples of code comments (with before and after) can be found in the Appendix A.1.

In addition to this, there is also the possibility to display a Copyright Notice in the first lines of every file along with an author reference. This small detail is a best practice that aligns with industry standards and serves multiple practical purposes such as clarifying ownership of the code, preventing copyright infringement and overall maintaining professionalism.

Code comments provide additional value to the code itself and help the flow of development by providing core information to the end user when needed.

4.3.3 Traceability

The implementation of traceability is related with the report feature, however, traceability can exist without a report and a report can exist without traceability 4.3.4. The main idea of traceability is to relate the coding files directly to the model so we are able to verify that a certain file came from a certain part of the model.

The base construction of the trace file was done in JSON, this choice came from the fact that JSON provides information while still maintaining a structure that can easily be ported elsewhere and can easily be machine readable.

This is an example of the JSON produced by a traceability analysis:

```
{
 "system": "Robot_Vision_I_Instance",
 "generatedAt": "2025-07-28 16:26:49",
 "location": "C:/Users/results",
 "components": {
```

```
"Usb_Cam_Nd": {
 "file": "robot_vision_i_instance/src/Usb_Cam_Nd_process.cpp",
 "type": "process",
 "subcomponent": [
 {
  "name": "Image_Broadcaster",
  "qualifiedName": "Robot_Vision_I_Instance::Usb_Cam_Nd::Image_Broadcaster",
  "codeTrace": "@trace name='Image_Broadcaster'",
  "line": "15",
  "type": "thread",
  "properties": {
   "Dispatch_Protocol": "",
   "Period": ""
  }
 },
 {
  "name": "UsbSpinner",
  "qualifiedName": "Robot_Vision_I_Instance::Usb_Cam_Nd::UsbSpinner",
  "codeTrace": "@trace name='UsbSpinner'",
  "line" : "20",
  "type": "thread",
  "properties": {
   "Dispatch_Protocol": "",
   "Period": ""
  }
 }
 ]
},(...)
```


As shown in the code above, the JSON relates the model to the code in various forms, mainly code traces, qualified names, file and package locations. On top of that, it provides additional info that is useful for both developers and end users, developers can easily use this JSON as input for various tools (generating an alternate report from the one provided for example) and users can use it to trace parts of the code dedicated to a specific component or subcomponent using a combination of the file locations, code traces and line number.

The way the line number is detected is quite complex in practice but more straightforward in theory:

- When traceability is enabled, the code generates with traces, those traces are not language or tool specific, hence they don't get picked up by other tools or as language syntax.

- The same traces are generated in the JSON.
- Using the JSON and the traces corresponding to sub components in the larger component (which is represented by a C++ file) the C++ files are scavenged for traces, which, when found, get saved directly below the *codeTrace*, as the line correspondent to it.

Using JSON simplified a lot the process of assigning the correct line to the correct trace since JSON can be manipulated with Java code in a simple and effective way.

In addition to this, traceability links were created using in the report (see Section 4.3.4) to ensure that the JSON file can be directly related to the C++ files using the *@trace* indicative.

Traceability ensures to the end user that the generated code came directly from the model that he chose, moreover it simplifies the search for a specific representation of a model component inside the code or vice versa.

4.3.4 Report

The report was implemented with the goal of providing a clear, concise and structured way of visualizing exactly how the model translates to code. To do this, the report is generated in HyperText Markup Language (HTML) using Acceleo. This is important since it overcomes a few constraints, mainly:

- **PDF with Java:** The major advantage that producing HTML reports with Acceleo fixes is that producing PDFs with Java code, although more practical[2] on the surface, it has many limitations when it comes to designing and structuring PDFs. In addition, the verbose is very unintuitive and can quickly grow in complexity.
- **Extensibility:** Since HTML is a simple and well known language, adding details and making changes is straightforward when comparing it to other tools like LaTeX. In this case, the usage of LaTeX isn't really justified since it is more directed to academic circumstances and not fast paced development.
- **Styling Issues:** Using raw Cascading Style Sheets (CSS) in combination with HTML is no doubt the best choice when it comes to styling the report. Java has many limitations when it comes to styling and LaTeX is very strict and error prone. For a basic report such as the one generated, CSS styling is more than enough.

Once these constraints were no longer an issue, the development of the report was much more intuitive. The base was made with HTML and CSS, which enabled a high degree of custom styles while also simplifying the integration of useful information.

Since the first iteration of the report was done in HTML, the transition to PDF was not needed since Eclipse naturally opens and displays HTML files in a selected browser. If

[2]Less technologies involved. Just Java as opposed to Java, Acceleo and HTML

there was, however, a need for an HTML to PDF transformation it could easily be done via workflow[3] after the PDF is generated.

The report has many features, its high customization enables the creation of different parts when needed, hence omitting redundant or unwanted sections according to user needs. One of the core elements is the ROS specific option, which highlights ROS specific components that were generated or ported from the model. This allows ROS developers to have a much clearer idea of how parts of the input model were represented in the ROS part of the code, for instance, what thread corresponds to a Publisher[4] or Subscriber[5].

The report takes a longer time of generation when comparing to other parts of the generation, this is because it also includes error reporting, which analyses every code file for errors.

This functionality is loosely tied with the report in order to better display the results of the Cppcheck tool, which was already explained in Section 4.3.5.1. However, since the output of Cppcheck is difficult for end users to interpret[6] the best way to display is using the report.

The contents of this report give a lot of insight on the generated code, they go into detail about the technologies, build system and models used to generate the code, giving the time of generation and the output folder as well. Along with this, a navigation section also gets generated which facilitates the finding of a specific component or section. Before the components, a short summary of the generation is displayed with info such as number of generated files, threads, code language and configurations applied. Every single process (originated from the AADL model) is documented in the components section, detailing the file they are located on and the threads managed by them, optionally a ROS process summary might be generated as well for each component.

In a way, the report serves as the bridge that unifies all the details of the project together in a condensed but clear manner, including: configuration used, source file location, content generated, and summaries of that content.

4.3.5 Code Quality Checker

A fine addition to the code generator was the introduction of a code quality checker powered by *Cppcheck*, which analyses every generated code file an points out errors immediately after generation. This allows the end user to quickly identify syntax errors and common code problems[7]

Moreover, the code can be analyses at any point, meaning that *Cppcheck* can verify the correctness of user generated code if needed. Providing value even after being used for

[3] More on workflows in Section 4.4.

[4] Publishers generate and publish data (sensor readings, etc).

[5] Subscribers listen for and process data sent by Publishers.

[6] It's not easy to read since it comes in XML format.

[7] This certainty is based on the capability that *Cppcheck* has to report code errors. It does not guarantee that the code is semantically functional, just syntactically correct.

its main purpose.

4.3.5.1 Cppcheck

Cppcheck is a static analysis tool for C and C++ code, it's designed to detect a wide range of programming errors, potential bugs, and code style issues without executing the program. *Cppcheck* analyzes the source code to identify problems such as memory leaks, uninitialized variables, unused functions, and violations of coding best practices. It's a lightweight, platform-independent, and highly configurable tool, which makes it suitable for integration into our code generation workflow.

Cppcheck has both an open source and a commercial version, in this case, the open source version was chosen due to RAMSES being in a development state. If the commercial version is deemed necessary, the change will not impact the current setup significantly as *Cppcheck* will only be stricter when it comes to code rule application, meaning that it will apply more rules on top of the current ones, generating a more enriched error report.

4.3.6 ROS and C++ separation

One of the underlining problems of the base generated code was the fact that ROS code and C++ code where closely coupled together. This is an issue since most of the C++ implementation is ROS independent, meaning that it could, in theory, be reused in contexts where ROS is not available or not needed. However, since the generated code is tight coupled with ROS, it forces the presence of ROS-specific dependencies even in parts of the system that do not require them.

To combat this, a new approach to code generation was implemented. The main goal was to port the ROS code inside the C++ files onto another folder, adjacent to the remaining C++ files. This can be better understood visually, we part from this:

```
src/
  Color_Tracker_process.cpp
  Image_Display_2_process.cpp
```

And we split the files into their ROS and C++ counterparts, like so:

```
src/
  impl/
   Color_Tracker_logic.cpp
   Image_Display_2_logic.cpp
  ros/
   Color_Tracker_node.cpp
   Image_Display_2_node.cpp
```

With the files now separated, it is possible to reuse the C++ code present in the impl folder without the ROS dependencies. This also opens more the window to the inclusion of company specific libraries that can, if needed, be split into ROS and C++ only counterparts.

To further improve on the ROS side of the customization, there is also the option to choose the thread executor for the given node[8]. There are three fundamental thread executors, they can be described as follows:

- The Single Threaded Executor processes all callbacks (from incoming messages, timers, services, and events) of one or more nodes in a single thread, ensuring that only one callback executes at a time.
- The Multi-Threaded Executor creates a configurable number of threads to allow for processing multiple messages or events in parallel.
- The Static Single-Threaded Executor optimizes the runtime costs for scanning the structure of a node in terms of subscriptions, timers, service servers, action servers, etc. It performs this scan only once when the node is added, while the other two executors regularly scan for such changes.

These three options fundamentally do not change how the generated runtime application works, but they give it different options when it comes to managing execution performance.

> Some complex applications might benefit from multi-threading while more simple ones can offer the fairly same performance in any threading type.

4.3.7 File Management

The intent of having a file management option in the configuration was to dynamically manage how the files are generated, including if they overwrite previously generated files.

The main addition however was the ability to add a header prefix to every generated code file, further facilitating copyright and licensing enforcement.

This file header can be any phrase or character but due to operating system restrictions, some character have to be filtered out, here is an example of this filtration:

```
..CON:my*file?<>|aux\0COM1  "

↓

CON_my_file____aux_0COM1___
```

[8]This also takes effect when decoupling is disabled.

By default, any forbidden character is replaced by an underline and spaces at the start and end of the word are trimmed. This eliminates eventual errors in the code execution.

A good example of the result of this file header implementation is, based of the files in Section 4.3.6, when adding the header *Nova"* becomes:

```
src/
  impl/
   Nova_Color_Tracker_logic.cpp
   Nova_Image_Display_2_logic.cpp
  ros/
   Nova_Color_Tracker_node.cpp
   Nova_Image_Display_2_node.cpp
```

Note that the apostrophe is transformed into an underline as mentioned previously.

The overriding system is another file management option. Previously, the code generator simply generated the files in any circumstance, effectively replacing every file, every time. In a simple model this is not an issue, however, in a larger model this can quickly become a performance bottleneck since it implies generating everything, every time.

When overriding is enabled, the generator does not change, it still generates every file every time, when it is disabled however, it only generates files that are not yet present. For instance, if we have a complete project, no new files will be generated or substituted, in contrast, if one file is missing, the generator will generate only the missing file.

Even preliminary tests produced a significant difference in execution time when overriding is disabled[9], proving that it is worth it to give the user the choice between generating the whole project or just specific parts of it.

4.3.8 Legacy Code Integration

The main problem that legacy code integration solves is the following:

There are C++ libraries that need to be used by specific ROS components, these libraries should be included in the ROS project when needed.

To put it simply, in order to produce the functionality that the input model entices, certain legacy libraries need to be included, not just in the project, but called in the code as well. The way it was done is clear:

- Firstly, libraries that the code will need, based on model components used, are listed.
- Then, the code for those libraries is copied or referenced in the project folder.
- Lastly, this code is called in certain places based on functional necessity.

[9] Up to 500ms less than the override enabled counterpart.

For this to work it is crucial to accurately detect which components already have libraries associated to them. This comes before the component is even instanciated, meaning: when we write the AADL system we want, the components used by that system already have libraries associated to them. This is done by explicitly including the libraries in the components properties.

The baseline code generator[10] already somewhat entailed to this functionality by not generating a skeleton when the library property was present, however, it lacked actually referencing that library properly and including it in the project.

The solution to this was straightforward: to include the libraries (when the configuration defined so) and their reference properly in the generated code.

The base for choosing the libraries is to analyze the properties of each process in the input model. The property *Source_text* reveals the library implemented by that process, meaning that it is not necessary to implement it again so we skip the code generation entirely for that specific process. In the end of the source code generation, the libraries needed by those projects are imported to the project and added to the */lib/* folder or referenced directly in the make file.

The reference mechanism is tied to the target operating system (Section 4.3.12) in the sense that the code referenced depends on the Operating System (OS) to establish a path. This fixes the issues with cross-platform development, for instance, it is possible to reference the libraries with a Linux path while using a Windows based machine.

This inclusion of the legacy code libraries was of extreme importance since it was one of the core goals of the thesis (see Section 1.3) and it's a crucial step to a complete ROS project generation.

4.3.9 Generation Hooks

The need to run certain scripts before or after project execution is very valuable. Either being quick cleanup scripts to prepare the project folder for a new version or post generation scripts to automatically test the generated project, these scripts can bring value and a higher of automation to any project.

For those reasons, generation hooks where included in the configuration language. The premise is fairly simple: to execute user created code before or after the actual code generation.

Even though that option is available in the configuration, in the specific case of RAMSES, there is no need to define a component that executes a script, meaning that there is already a way to execute terminal based commands in the project. The definition was still kept in the configuration language although it was not implemented in practice due to a more practical option that is depicted in Section 4.4.3.

[10]The non configurable version of the code generator, done by António.

4.3.10 Multiple Configurations

When testing the previously finished features, a few issues stood out:

- The difficulty to quickly change from one configuration to the other.
- Handling multiple configurations at the same time.

The main problem is that, if we want to create and customize a new configuration we have to start from scratch and use a new file. To combat this issue, a new way of creating configurations was introduced: the list of configurations.

The premise is that instead of having multiple files, each one representing a different coding language, there is a single file with a list of configurations.

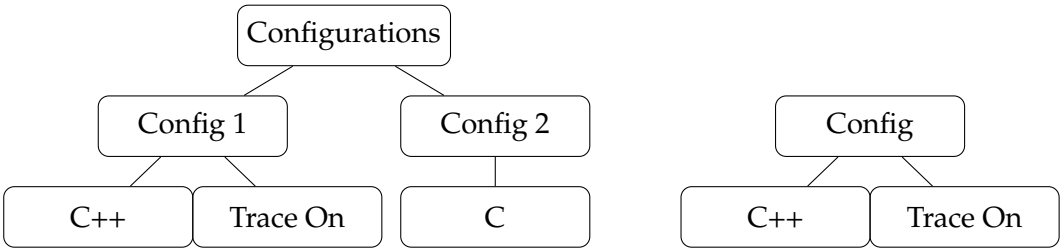

Figure 4.2: Comparison between multiple configurations (left) and a single configuration (right).

This way, instead of handling multiple files we can, like illustrated in Figure 4.2, manage every single configuration in one file and the same configuration can be used by different generators since each one can find the configuration it needs.

This is an addition, not a restriction, so, as we will explore more in the next section 4.4, we can still use a single configuration as input or a list of configurations, further improving the flexibility of the code generator.

Even though this list addition works well with the base idea of the configuration, a different issue halted the use of single file configurations entirely: the dynamic addition of programming languages.

4.3.10.1 Multiple Languages

A key contribution of the DSL is the dynamical addition of multiple distinct configurations, however, as the language is implemented into different projects, an issue arises: adding more programming languages.

At face value the solution to this is simple[11] but intrinsically its more complex and involves modifying the current metamodel.

Figure 4.3 represents the list of languages before the change. Since the languages are represented by an enumeration, the addition of a new language involves coding and regenerating the configuration model.

[11] To add to the already existing enumerator.

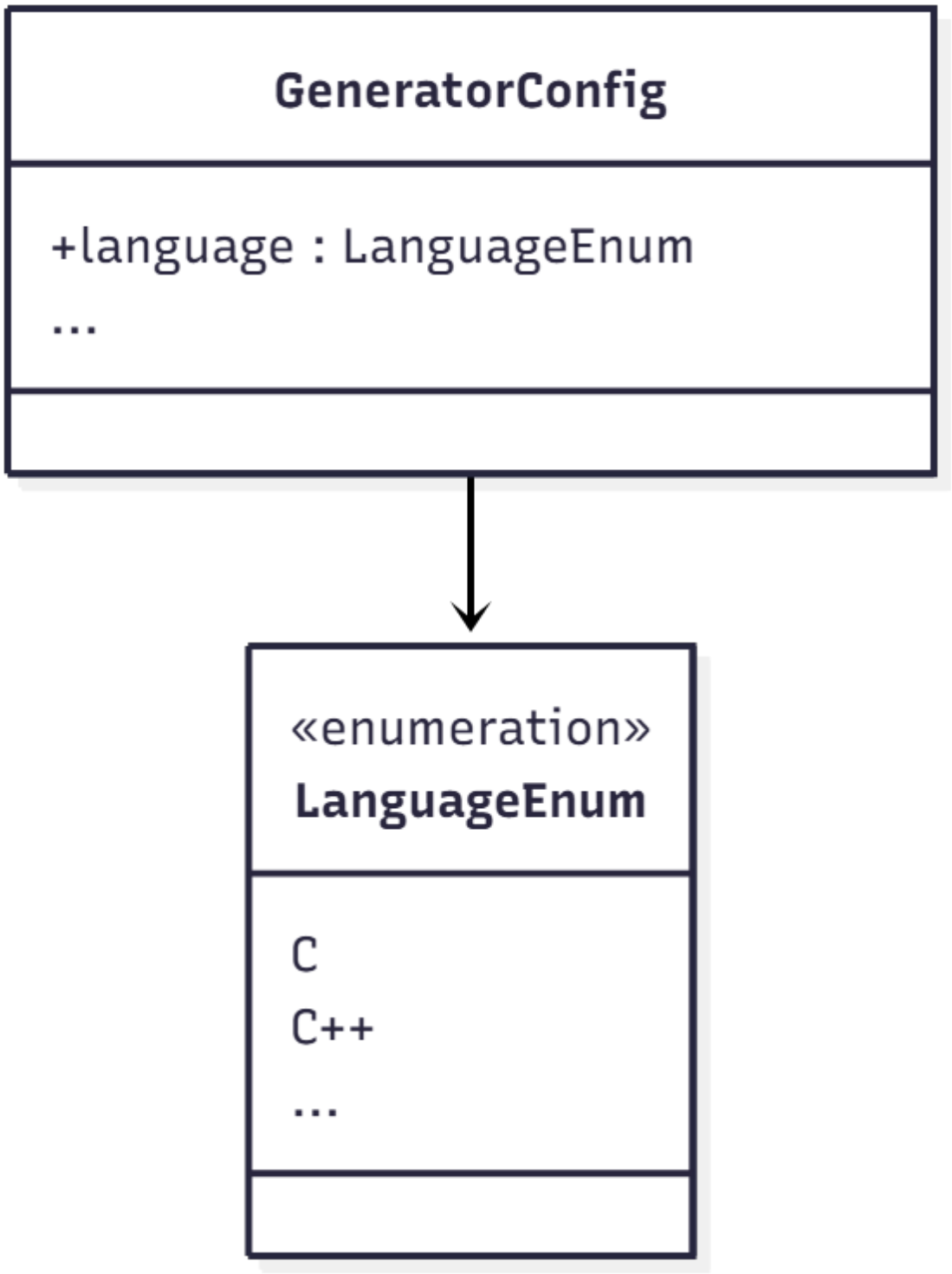

Figure 4.3: Class diagram representing the base idea of choosing a coding language

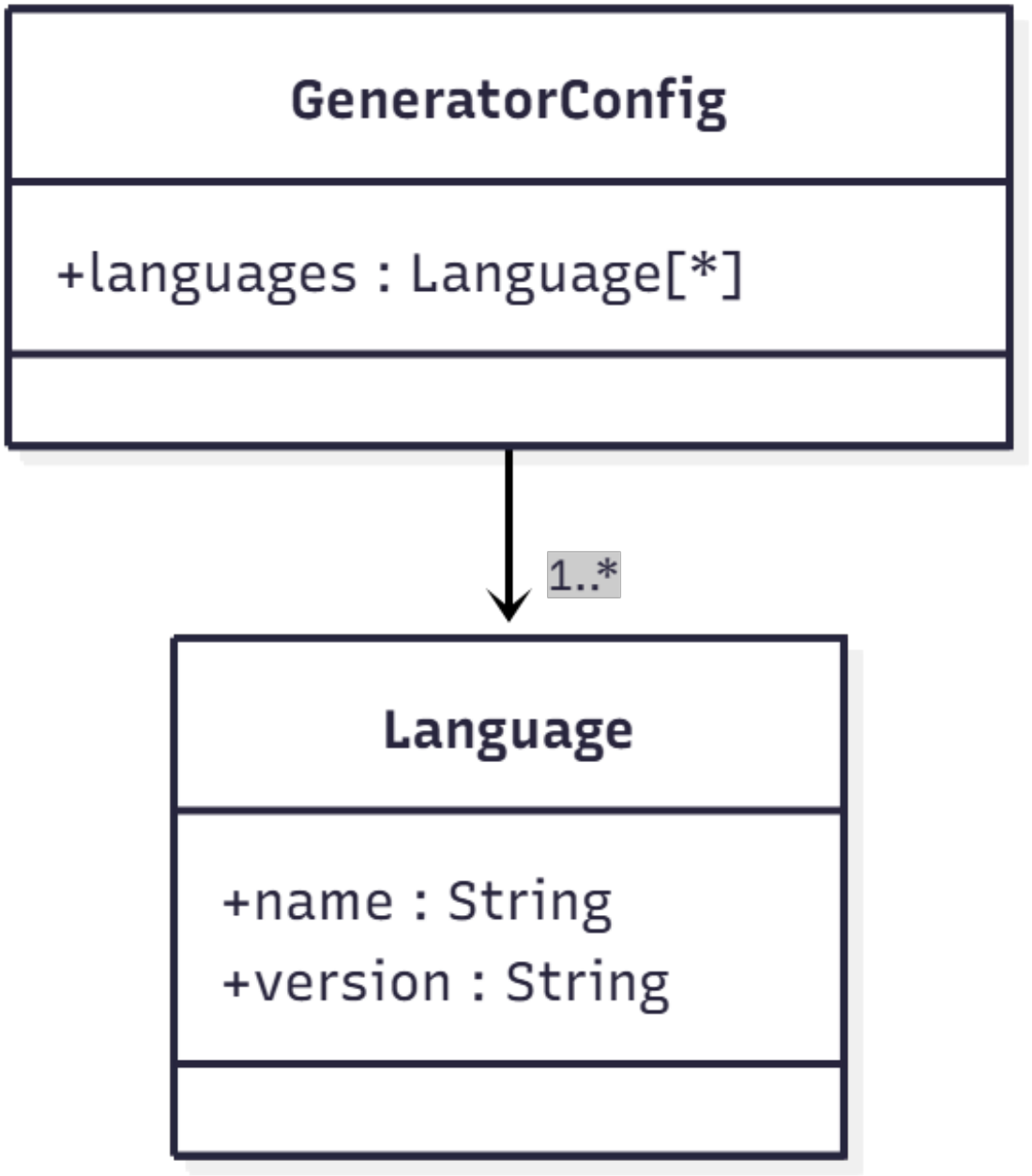

Figure 4.4: New, improved class diagram representing the selection a coding language

As we can see in Figure 4.4, since a coding language is now represented by a class, it is now possible to add new languages without modifying the metamodel, which makes the process more seemless and able to be automated so the end user can add their desired coding languages with a simple menu instead of having to write code.

The DSL architecture suffered a tiny shift which ended up changing the way single file configurations work. Since now a GeneratorConfig requires a reference to a Language, this Language needs to be imported via *Load Resource...* to be associated with the configuration. This of course, only applies to single configuration instances since the multiple configuration architecture has a list of coding languages.

In simpler terms: the programming Language option cannot be "inside" a configuration because it will be associated with that configuration. Language options need to be in the same level as configurations in order to both be created dynamically and not depend on one another.

4.3.11 Build Integration

To further improve the control the user has over the generation, the option of build integration essentially allows the deactivation of the *source*, *make*, *package* and *launch* files. Since those files were always generated by default, now it is possible to generate a completely empty project (for whatever reason) or simply isolate parts of the project to find critical errors both in development and when testing.

Although simple, this addition is very useful and, once again, adds to customization and control over the generated files. By default, the files previously mentioned are set to generate.

4.3.12 Target Operating System

Since there is the possibility to want the generated project to be produced for a different operating system than the current one the user uses, instead of detecting the current OS and assuming it as the target OS, the configuration language has the option to produce the generated code adapted to either one of the three main operating systems: Linux, Windows and MacOS.

This provides a simple but effective way to ensure portability throughout the different OSs. The differences in the generated code are mainly the way to reference files and the way the make file is generated.

4.4 Workflows

The process of generating files involves feeding both a configuration and input model to the generator, which then processes this into C++ code, an HTML report or a traceability

JSON. There are many issues that arise from the way the generation process works, mainly that:

- Many Java main classes have to exist.
- In order to generate everything needed to run a ROS project many Java methods have to be run.
- Some classes need to be ran after, and only after others.
- These methods cannot be run in a single click, as that would require a complete redesign of the current architecture making it less flexible.

The solution for these issue is workflows. Workflows allow the automation of certain tasks (reading, writing, or transforming models) by plugging together several workflow components, which are executed consecutively. A workflow component performs a certain task and can be configured with parameters.

This allows for the execution of specific parts of the code generation in an isolated way, as well as the execution of every part needed to produce a complete and functional ROS project, on top of that, JUnit tests based on workflows are also possible and already a common pattern on the RAMSES project.

Even though workflows are already used in RAMSES, there was no concrete workflow to execute the code generation in the way it is done so far. To solve this problem, a new custom workflow component had to be developed.

4.4.1 Acceleo Workflow Component

The custom workflow component created was the AcceleoGeneratorComponent, which simply parsed all the inputs (AADL input model, configuration and output folder) and executed the Java class responsible for the specific code generation.

The beauty of this component is that it is optimized to every single Acceleo Java class present in RAMSES, meaning that, in the workflow, every Java class can be represented by a AcceleoGeneratorComponent which facilitates the sequential execution and removes the need to have components specific to a certain class.

In addition to solving the previously mentioned issues, this new component was also engineered to run functions before and after generation if those functions existed in the input class, as an example, an input template A (with an associated class A) can have as methods *shouldRun()*, *preGeneration()* and *postGeneration()* on top of the generic method *doGenerate()*. These methods have key functions that help enrich the component beyond simply executing a class, namely:

- *shouldRun()*: if true continues with the code generation as default, aborts generation otherwise.

- *preGeneration()*: executes preparations that a given template might need for generation, for example, providing additional generation arguments.

- *postGeneration()*: executes post generation code, for instance, filling traces with the code lines they generated in.

It is worth to note that the *preGeneration()* and *postGeneration()* functions are similar to generation hooks (see Section 4.3.9), however, these functions are executed at class level while generation hooks execute before or after the whole code generation process.

In addition to all of these features, this custom component also simplifies changing the inputs of the generation. Sinse there is a property for each of the inputs in the workflow (AADL model, configuration model, working directory and output folder), the change in the AcceleoGeneratorComponents is simple: a change to the property will affect all the workflow components that use that property.

4.4.2 Dynamic Acceleo Workflow Executor

The custom component described in the previous Section 4.4.1 works flawlessly for its main purpose, however, a key issue arose: instead of feeding the template to the model, we feed the class that that template represents. This is an issue since if we want to execute a specific Acceleo template we need its class.

The solution to this problem was clear: instead of calling the classes independently, the component would call a generic class that would handle any Acceleo template. This did not come without drawbacks, mainly that handling every template the same way reduced their overall specialty. This was an issue that the previous component solved by integrating additional functionality in each individual class as needed.

Essentially the two architecture choices where either full specialization but low integration capabilities or high integration and low specialization. However, there was also a hidden third option: both.

A new version of the Acceleo custom Workflow component was devised, the premise was that the component would receive the template name (not the class name) and search for the class represented by the given component, if none was found the component would fallback to the generic Acceleo generator class. This hybrid architecture means that, when feeding a template name that does not have a Java class associated, the Workflow will execute it seamlessly, provided it has an AADL model and a configuration file. Similarly, if there is the need for a more in depth execution, a Java implementation of the template can still be done.

4.4.3 Script Workflow Executor

One of the features mentioned both in the feature model 2.1 and in the client meeting 4.1 was the ability to execute pre or post generation hooks as discussed in Section 4.3.9.

This functionality was primarily aimed to be present in the DSL as a way to execute Java code to consequently run a command script, however, in the very specific case of the RAMSES environment, a different option is more viable and already implemented: System Command Component.

The System Command Components allows for the execution of commands in a terminal (in this case a Windows terminal) by passing values in its Workflow component properties.

When running a script before code generation occurs, we need the following:

- **Arguments:** \c, pregen.bat[12]
- **Command:** cmd
- **Working Directory:** file:/C:/Users/user/robot_vision_i_instance

With this information, the SystemCommandComponent runs a terminal on the given directory that executes the command *cmd \c pregen.bat*, effectively executing the file *pregen.bat*.

4.4.4 Execution Times

With the workflow built and running, it was possible to get an idea of the execution times of each part of the workflow.

4.4.4.1 First Workflow Execution Time Test

To perform the first volley of workflow execution time tests, a functional AADL model was picked due to its correctness and previous usage in some examples. The configuration was also tuned to its maxed settings, meaning that every option was checked (report generation, traceability, etc) so that it would produce a better range of timetables, the complete configuration details for the this context can be found in A.4.

In this phase, 15 execution runs where documented, which can be found in the Appendix A.2.

Taking a look at Figure 4.5 we can see that naturally some parts of the workflow take more time than others, mainly the code generation and the report generation.

The Launch, CMake and Package files take much less time than the other ones due to their simplicity and to the fact that they generate a single file, as opposed to the Code generation which, in this case, generates 7 C++ code files. The report takes even longer than that because, on top of documenting the generated files and build an HTML report, it also checks those files for code quality issues, which ends up taking even more time. The JSON Trace file is pretty straightforward, however, since it scavenges the code files for the exact placement of the traces, it does end up with an execution time bigger than the other single file generations.

[12]This represents executable that we want to run.

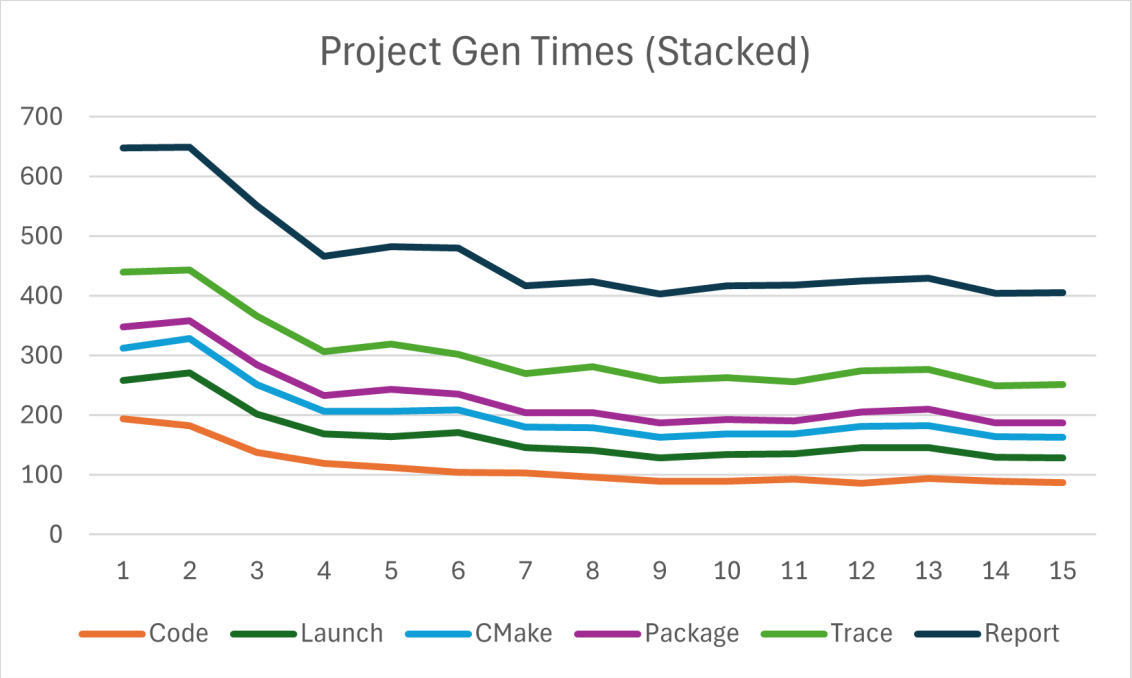

Figure 4.5: Code generation execution times

Since the base project did not have a workflow that successfully implemented the ancient Java code, it is not possible to directly compare the execution times of the old version of the code generation with the new one. However, we can estimate the time it took to execute a project generation in the ancient generator with the information from Table A.3, which, after measuring the time it takes to generate the Launch, CMake, Package and C++ files, comes around to 231,6ms, essentially half of the current average execution time of 479,9ms. This is expected since both the Trace and the Report take a significant amount of time. This interval can be even bigger if we consider that the original code generator did not even have a configuration file.

As mentioned previously, Figure 4.6 confirms that the average execution time seems to have doubled with the introduction of more processing heavy modules (Trace and Report), this is not a big issue as the total execution time is still below 500ms which is still very acceptable for the current conditions and given the huge amount of functionality and dynamicity added to the code generator.

All in all, workflows proved to be an essential part of the code generation pipeline that tie together the functionality of every individual configuration option in a single, harmonized execution. In practice however, the editing of the configuration and the execution of the workflow will not be done in the raw files, but in Osate using buttons and preferences for execution and configuration respectively.

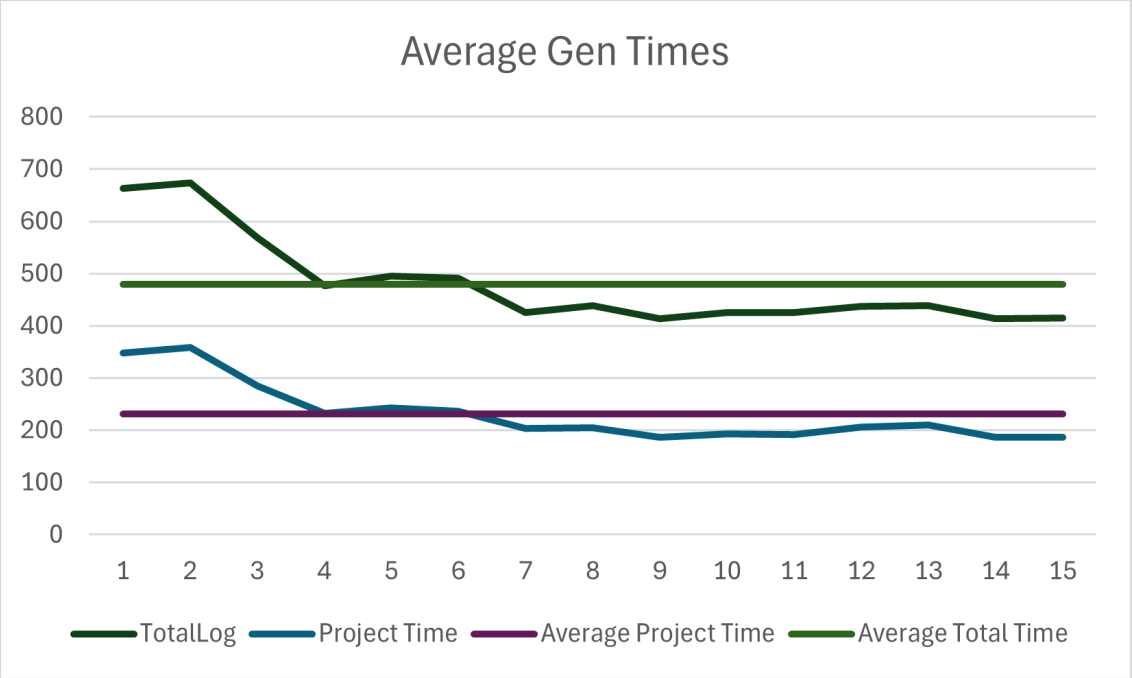

Figure 4.6: Average code generation execution times

4.5 Domain Specific Language

In order to customize the output of RAMSES's code generation, a DSL was created using the Ecore framework. Which enabled a clean and straightforward way of defining configuration options based on code generation needs.

The configuration language as a base can be used in two formats:

- **Single configuration:** Configuration that allows the selection of language, naming convention, report options, architecture choice, file management, code comment management, traceability options and generation hooks.
- **List of configurations:** Dynamic list of configurations that allow for quick change and assignment of configurations when used.

As specified in Section 4.3.10, this choice is highly beneficial since it provides more flexibility to the usage of the configuration language itself with no notable drawbacks.

Going more in depth, the DSL offers specific options given each topic, notably:

- **Language**[13]**:** C++ or C.
- **Naming Convention**[14]**:** Style (UPPER_CASE, lower_case, etc), affix and affix position.

[13]Currently only 2 options, will be expanded in the future.
[14]Derived directly from Section 2.4.1.1

- **Report:** Generate (whether the report gets generated or not), include traceability, include summary tables, include logic summaries (in the case of RAMSES, ROS summaries), show detected errors and open automatically.
- **Business Logic:** Decouple (separate from the code) and executor choice (Single executor, multiple or single static executor).
- **Files:** Overwrite existing, output directory and header prefix.
- **Code comments:** Enable, copyright notice and author.
- **Traceability:** Enable and produce diagram.
- **Extension:** Pre and post generation scripts.
- **Build Integration:** Control over generated files.

These options control each part of the generated project and together they offer a high degree of customization to the generated code. The effective way of using the configuration is via the preferences window on Eclipse, where every option of every category is present in a different page.

DSL = "the configuration schema and semantics". It's independent from but used by RAMSES.

4.5.1 Custom Preference Store Component

Saving user preferences in the Eclipse workspace is a simple but effective task that allows for a direct way for the user to configure specific options.

In the case of the code configuration DSL, this preference window fits perfectly with the language. It provides a way for users to modify the options freely while also restricting and minimizing possible errors since the User Interface (UI) presented has guardrails that prevent the user from introducing errors into the DSL.

This approach for the UI however came at a cost: the storage of values was already implemented by eclipse, meaning that, by default, it was not possible to save the DSL metamodel whole, just a key:value map.

To counteract this problem, a new way of storing configuration values was developed. By implementing the Preference Store interface we were able to develop a new way of preference storage that had at its core the configuration metamodel.

This essentially means that the manipulation of the very base of the DSL itself can now be saved in Eclipse's preferences, allowing it to be used instantly.

In short, instead of having to apply preference values to the metamodel, we save the preferences directly to a central metamodel, cutting the unnecessary middleman (default preferences).

This model then can be used as the input for Workflows (Section 4.4) to generate code based on its configuration. Effectively giving the user control over the base metamodel while also restricting his possibilities to commit errors.

This storage approach does not affect the UI but it does simplify and improve the overall process of managing the DSL metamodel.

4.5.2 Main Options

At the core of the options for code generation, the basics are:

- **Output folder**
- **Target Language**
- **Target Operating System**

These options are strictly necessary for code generation to take place, hence being the main options. All options presented in this Section 4.5 are available to be configured under *Osate* > *Window* > *Preferences* > *CodeGen Config* in the Osate platform.

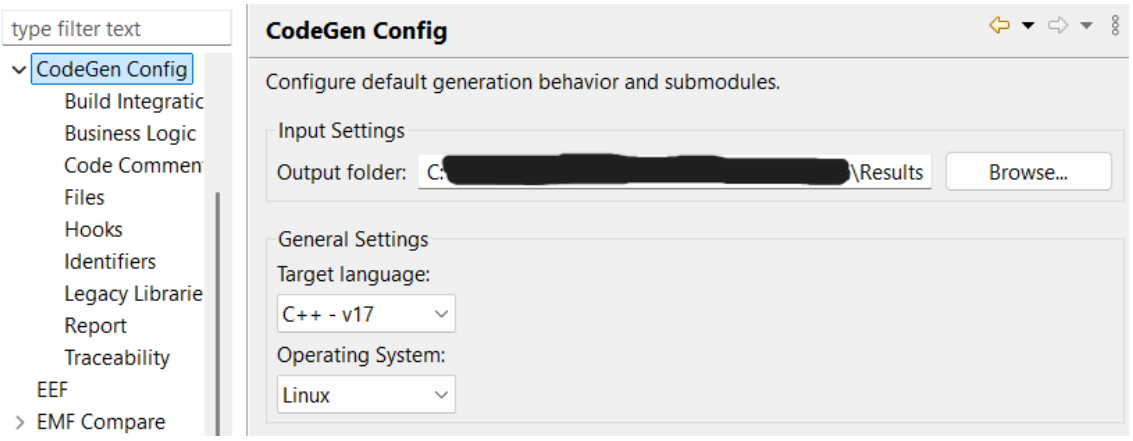

Figure 4.7: Main page of the code generation configuration

Figure 4.7 highlights how the main configurations appear in the preferences window. From here it is possible to select the output folder and the target language that will be fed to the generator for code generation.

The remaining sub-sections of this chapter describe configuration options that refine or extend the behavior defined by these core parameters.

4.5.3 Language

The language option defines the targeted language for the current customization. The language option also serves as a distinction between different configurations in the configuration list, allowing for the quicker identification of the selected language in the generation.

This option also serves to map the code Identifiers to their language specific casing[15] as explained in Section 4.5.3.

This option is highly extensible and, as RAMSES grows, can be extended with many other languages such as Java, Assembly, Kotlin, etc.

This option appears in the main page of the configurations as seen in Section 4.5.2

4.5.4 Naming Convention

The essence of the naming convention option is simple: to automatically change the casing on Identifiers to ensure consistency across all generated artifacts. It allows the user to enforce casing rules on classes, methods, variables, and constants throughout source files.

4.5.4.1 Identifier Style

The format on the style is done by analyzing the selected option in the configuration and then applying the style that corresponds to that option before code generation takes place. On top of the code, this also applies to every other instance of that identifier (CMake files for example), ensuring that every reference to that Identifier still preserves the style.

Most styles simply change every Identifier to the same style, examples of this are *snake_case*, *PascalCase*, *camelCase* and many others. The *DEFAULT* option leaves identifiers exactly as they have been written in the model, allowing for an effective nullification of the formatting process.

The *STANDARD* naming convention differs from the others as it allows for a dynamic Identifier naming, for instance: it allows *Classes* to be *Pascal_Case* while simultaneously having *Variables* in *snake_case*. This is very useful for specific restrictions where languages need different casing based on Identifier (Classes, methods, variables, constants, etc). This specific casing is very configurable and can be adapted to different programming languages, further improving the adaptability of the DSL.

4.5.4.2 Identifier Affix

Users may additionally define a prefix or suffix to be applied to all Identifiers

Moreover, the affix is not affected by the styling format, this is done to give more choice to the end user, if we wish to have the affix not follow the selected casing that is the default behavior. On the other hand, if we do wish the affix to follow certain casing the solution is to simply write it according to that casing.

All the options in the naming convention section are available to be modified in the preferences window as shown in Figure 4.8.

Enforcing identifier casing is especially important in parts of the toolchain where naming conventions may be interpreted semantically, such as ROS message fields and

[15]This only applies when the naming convention is set to STANDARD.

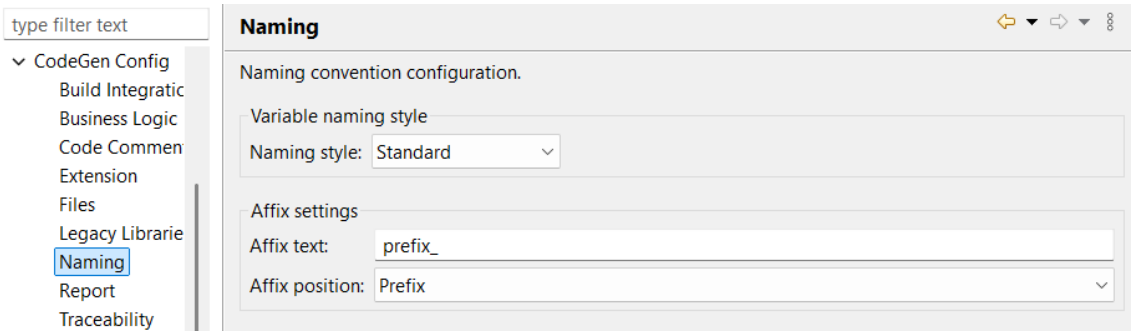

Figure 4.8: Main page of the code generation configuration

class names. Having the style configuration in the DSL ensures that all generated artifacts are syntactically consistent with the selected target environment.

4.5.5 Report

The report generation option offers various ways to customize the generated report that complements the code. This report is optional and, since it does take a good bit of time to be generated, can be turned off when not needed.

Other customization options include the logic summaries and summary tables, which add condensed info to the end of certain parts of the report. Traceability, when enabled, will include direct links to components, threads and models in the report.

Along with these options there's also the option to detect errors. If this option is toggled off the analysis will not be performed.

Similar to the other options, this option can be modified in the preferences as shown in Figure 4.9.

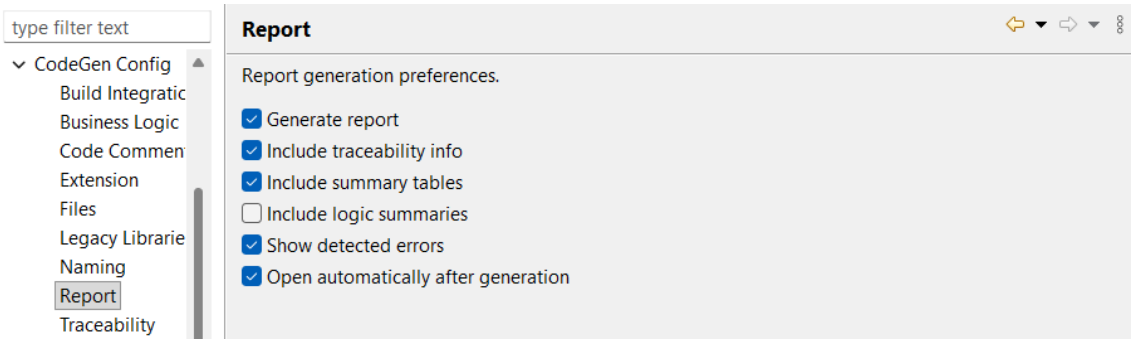

Figure 4.9: Main page of the report generation configuration

One key detail of the report configuration is that if the report generation itself is disabled (first option in Figure 4.9), the remaining options don't alter anything, meaning that it would not matter if, for instance, *"Include logic summaries"* was enabled or not since the outcome would be the same. For this reason, if the *"Generate report"* option is disabled,

the other options will be unmodifiable to convey to the user that those options only take effect if the first option is enabled. This behaviour can be seen in Figure 4.10.

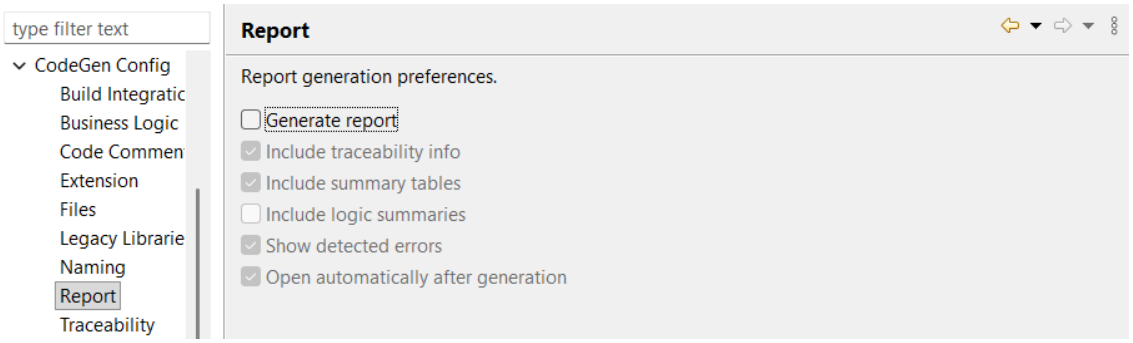

Figure 4.10: Report generation disabled

The report serves as a form of documentation of the generated system, allowing developers to verify many properties such as scheduling, port bindings and thread structure without inspecting the generated code manually.

4.5.6 Business Logic

The business logic options handle the technology specific part of the generation, in this case, its ROS.

The decoupling option splits the C++ code from the ROS specific code with a simple toggle, while the executor option offers a predefined choice between the thread executors used in runtime.

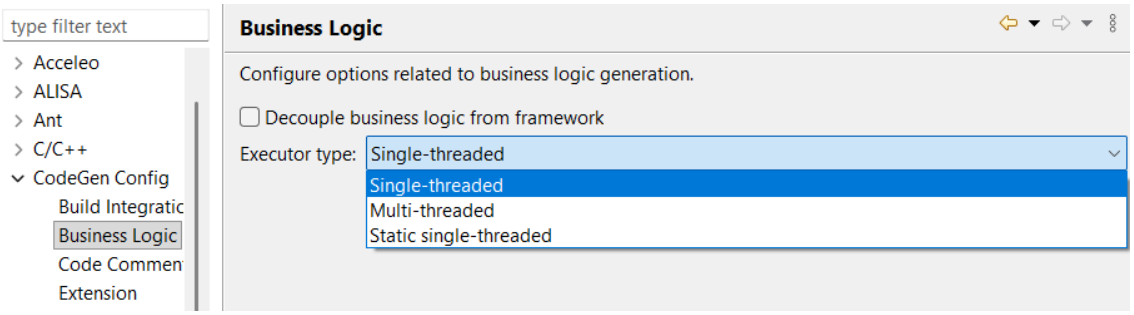

Figure 4.11: Page of the business logic configuration

Figure 4.11 highlights the business logic menu options.

These options isolate the platform-dependent runtime semantics (in this case the ROS node execution) from the platform-independent logic (C++ code). This separation enables the same architectural model to be plugged into different execution frameworks with small changes.

4.5.7 Files

The file configuration controls file naming and output directory structure. Users can specify a shared filename prefix and enable or disable overwriting of existing files. This is important when regenerating code in incremental development workflows.

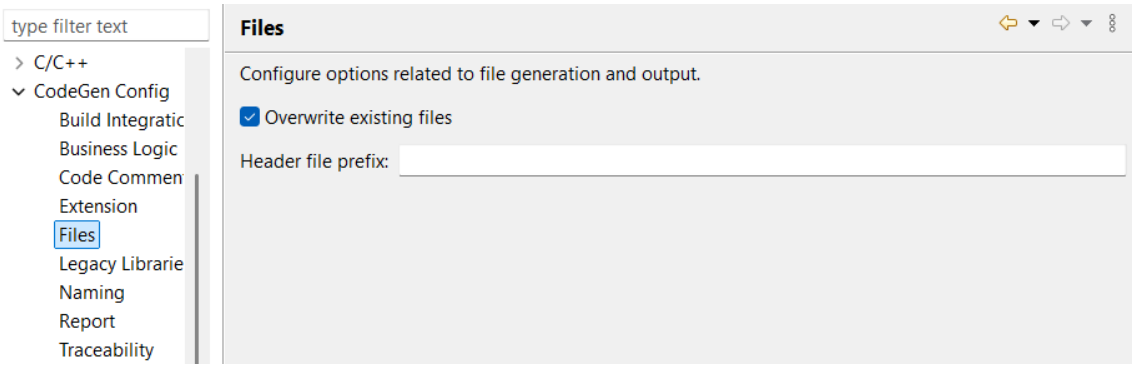

Figure 4.12: Page of the file management configuration

Figure 4.12 shows the options that control the file management.

4.5.8 Comments

In the comment options we can input both the author and a copyright notice for all code files. Documentation comments can also be enabled/disabled entirely.

By default, no comments are emitted.

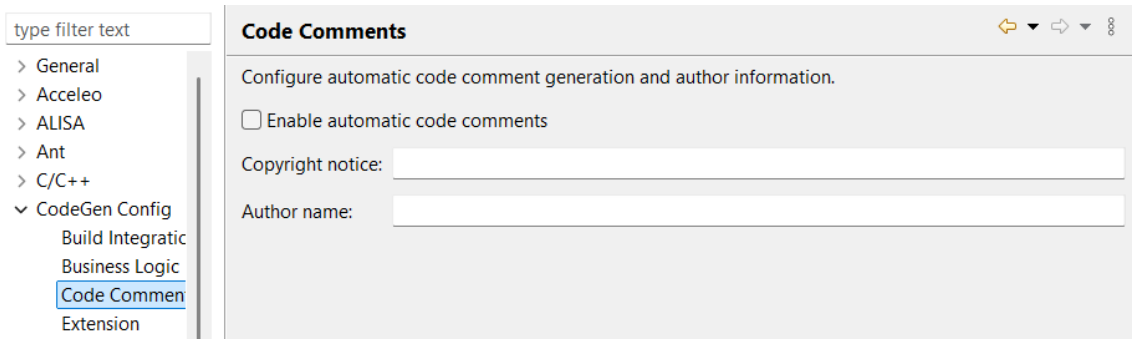

Figure 4.13: Page of the code comments configuration

Figure 4.13 illustrates the different code comment generation options.

4.5.9 Traceability

Traceability options enable or disable the generation of structured trace metadata linking model elements to corresponding code fragments. The JSON trace file may be generated independently, while diagram export is reserved for future work.

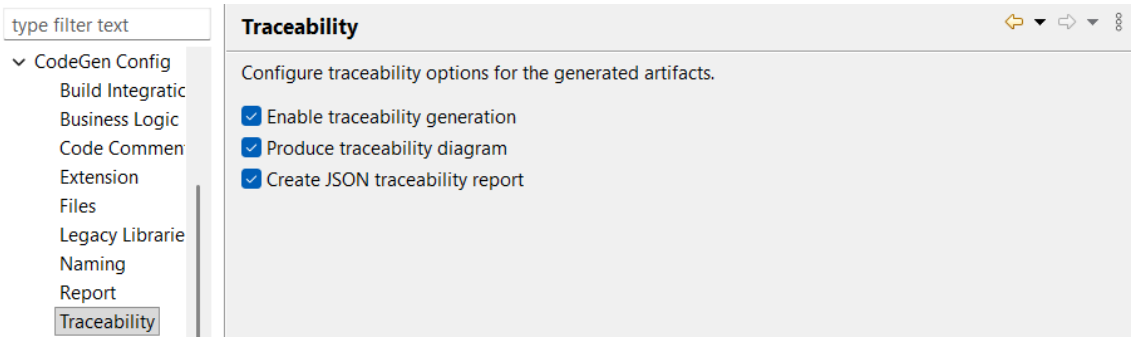

Figure 4.14: Page of the traceability configuration

Figure 4.14 shows the different options for traceability, as we saw previously with the case of the report generation (Section 4.5.5), if traceability generation is disabled, dependent settings become inactive. This behavior can be seen in Figure 4.15

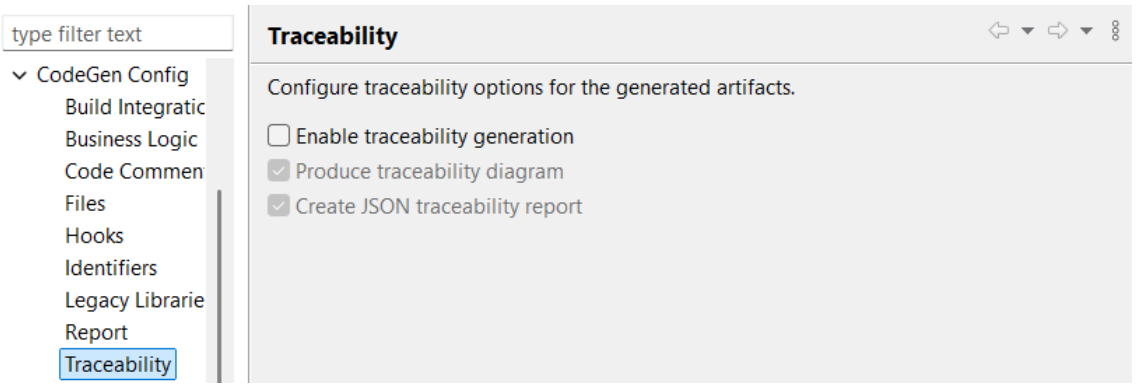

Figure 4.15: Page of the traceability configuration with traceability disabled

4.5.10 Extension

The extension option represents the generation hooks (see Section 4.3.9) which were implemented with an already existing workflow component. This settings configure pre and post generation hooks that allow execution of external scripts.

These hooks enable automation without modifying the code generator.

4.5.11 Build Integration

Build integration controls whether main project artifacts such as *source*, *make*, *package* and *launch* files are generated.

Figure 4.17 illustrates how the options can be selected. Each one of the options is independent from each other, meaning that they can be ticked on/off in any combination.

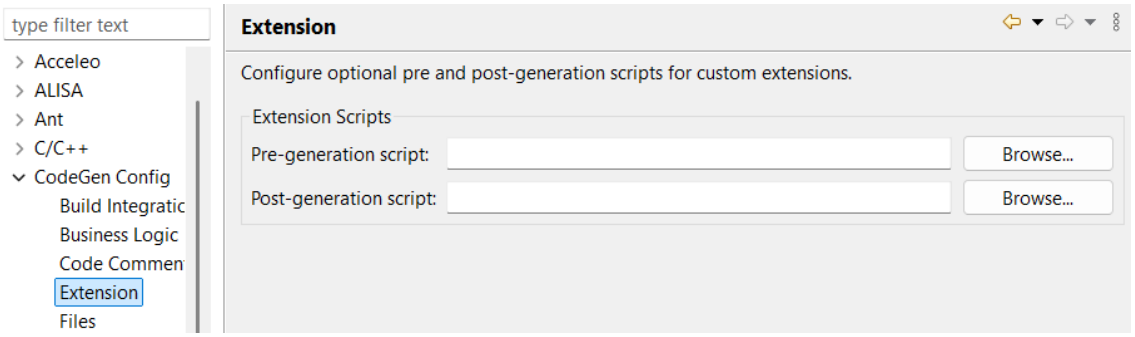

Figure 4.16: Page of generation hooks for code generation

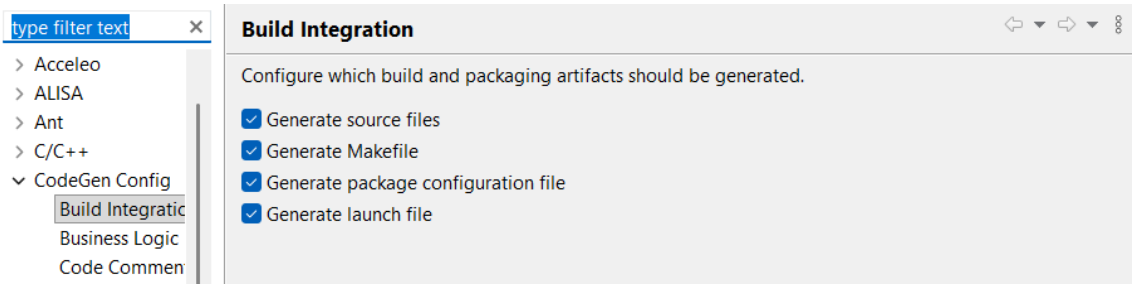

Figure 4.17: Page of the build configuration

4.5.12 Legacy Libraries

Legacy library handling determines whether external libraries are either included or not and when included can be referenced or copied.

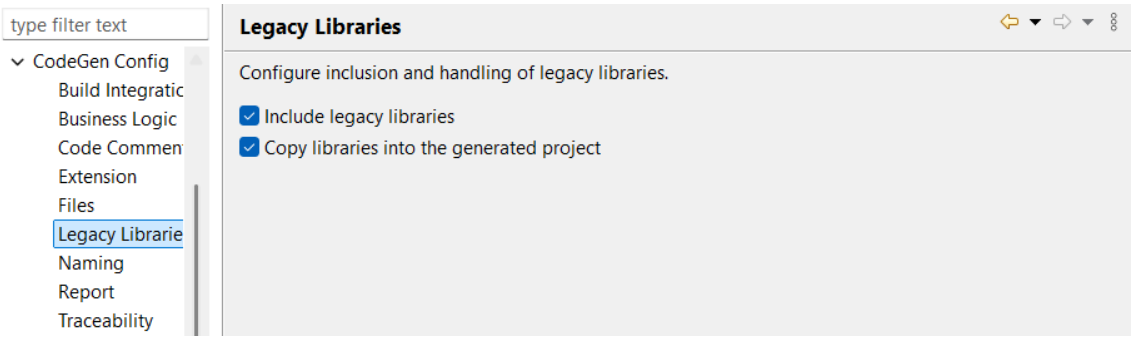

Figure 4.18: Page of the legacy configuration

Like other previous pages, the legacy page requires the inclusion of legacy libraries for the copy option to be enabled. Figure 4.18 illustrates how the page looks with all the options enabled and Figure 4.19 shows how the the copy option depends on enabling library inclusion, ensuring configuration consistency.

The ability to either reference or copy legacy libraries supports two distinct workflows: systems where libraries are centrally maintained, and isolated embedded deployments where all dependencies must be imported directly to the projects.

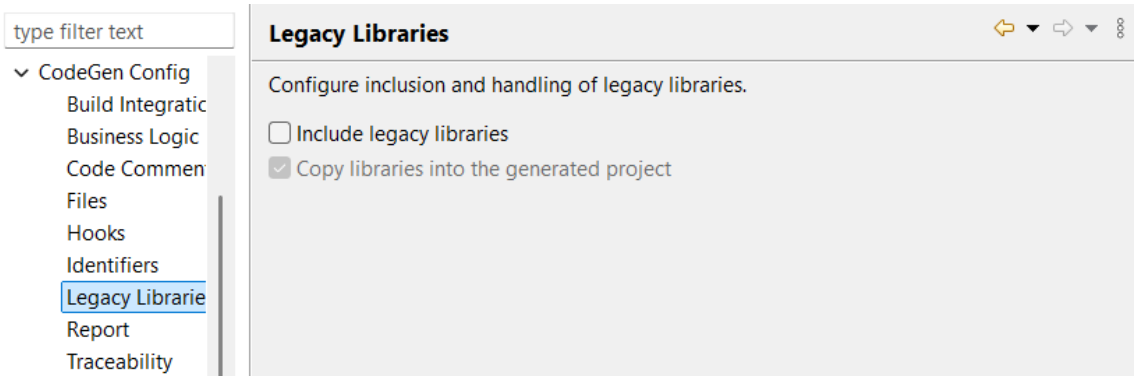

Figure 4.19: Page of the legacy configuration with library inclusion disabled

4.5.13 Target Operating System

This dropdown selects platform-specific adaptations to the generated code (for instance, file paths). Supported systems are Linux, Windows, and macOS.

This option is one of the core options and can be found in Figure 4.7.

5

Software Testing

This chapter documents the software testing done throughout the development of the thesis. It highlights how the specific tests where done, in what context and their results.

Software testing was conducted specifically for the configuration of the code generation part of RAMSES. This means that the goal was to ensure that the newly developed configuration options did not modify the runtime results of the generated software, hence maintaining the core goal of the input model.

5.1 Integration Testing

This type of testing is recurrent along the development, it serves to ensure that the features implemented are functioning as intended and that the previous version of the software has retained its functionality when new features were added.

Integration testing was performed in two clear stages, matching the incremental development of the configuration language. The first phase, in August, ensured that the addition of configuration features did not change the behavior of the baseline generator. The second phase, in October, focused on the validation of the more advanced architectural and runtime-related configuration options and refinement of the testing methodology to better reflect the intended modularity of RAMSES.

5.1.1 Integration Testing: August

After the development of the first batch of features, the testing served to ensure that the previous version of the code generator was still working as intended and to validate the integration of the code generator configurator in the pipeline.

The main testing environment for the tests in this sub section was the following:

- Eclipse version 2024-03 (4.31)
- OSATE2 version 2.16.0.vfinal

- RAMSES2 version Aug 26, 2025 (commit 335579b6e58a188475e7a5e7ed099b9a549bfb25)
- ROS 2 Humble Hawksbill
- Acceleo version 3.7.15.202402190857
- MDELab Workflow version 1.9.1.202402292002
- EMF - Eclipse Modeling Framework SDK version 2.37.0.v20240203-1010
- Ubuntu version 22.04.5 LTS

The regression tests applied at this time were limited since the previous version of the software did not include any code generation customization, only the simple transition from model to code (M2T), which, after generating a ROS project in both the previous code generator and the at the time current one (with the first batch of features implemented), it was possible to verify that both generators still produced the same practical output, essentially, the code produced by both generators yields the same functionality.

The integration test part was done much more in depth since the goal was to intertwine the different new features and find out if they worked in perfect harmony as expected. The tests present in Table 5.1 were conducted.

Table 5.1: Feature dependency table

Test Case ID	Feature	Description	Expected Outcome
TC-INT-01	Identifier Management	Check that Identifier config is applied in code, traceability, and report.	Same naming convention is visible in all outputs.
TC-INT-02	Comments	Verify comment style and formatting are preserved in all files.	All comments retain exact style and placement.
TC-INT-03	Traceability	Ensure model elements map to the correct code location and report entry.	JSON and HTML show accurate paths and line numbers.
TC-INT-04	Report Generation	Ensure generation metrics are passed from Java generator to report.	Report displays correct counts and settings.

TC-INT-05	Code Quality & Report Sync	Verify quality checker warnings appear in both XML and report.	Both outputs show the same warning details.
TC-INT-07	Cross-Feature Stability	Test large models with custom Identifiers, comments, and style violations.	All features work together without breaking.

After conducting these tests, the following was concluded:

- **TC-INT-01 - Identifier Consistency:** Verified that the Identifier configuration set in the `config.generator` model is applied consistently in generated C++ code and reflected in the traceability file as well as in the report.
- **TC-INT-02 - Comments:** Generated code containing both single-line and multi-line comments, verified that:
  1. Acceleo templates correctly output the comments.
  2. Java generator preserves formatting.
- **TC-INT-03 - Traceability Link Accuracy:** Confirmed that model elements are mapped to the correct files and line numbers in generated code, and that these mappings are:
  1. Present in the JSON traceability output.
  2. Referenced in the HTML report.
- **TC-INT-04 - Report Generation Data Flow:** Metrics collected during generation (number of components, code quality warnings) are passed to the Acceleo generator rendered correctly in the report.
- **TC-INT-05 - Code Quality & Report Synchronization:** Triggered a known style violation in the model, generated code, and verified that:
  1. The code quality checker detects it and outputs XML accurately referencing it.
  2. The Java generator correctly embeds this warning into the final HTML report.
- **TC-INT-07 - Cross-Feature Stability:** Used a large, complex model with:
  - Custom Identifiers
  - Extensive comments
  - Known style violations

  Verified that all features (Identifier config, comments, traceability, reporting, quality checking) work together without breaking any module.

5.1.2 Integration Testing: October

This latter stage of integration testing concentrated on the more architecture and runtime-oriented features of configuration, as well as revising the way in which testing was performed in order to better align with the modular design principles of RAMSES. While tests in August concentrated on baseline correctness and compatibility, tests in October focused on verifying that these advanced configuration options (business logic decoupling, executor selection, file management, and build behavior) interoperate correctly and produce expected behavior when combined.

The testing environment remained mostly the same, with the exception of the execution platform: testing was moved from a physical Ubuntu installation on Windows to a dedicated Linux virtual machine. The RAMSES version was also updated to Oct 27, 2025 (commit 3eb5118b57903cc5077ad4218fc83e182cbab199).

The initial tests were conducted with a "parameterized" way in mind, meaning that the architecture prioritized code efficiency albeit with an addition of complexity. This in of itself is not an issue at all, the tests were still very efficient, accurate and highly maintainable[1]. However, the current RAMSES architecture benefited more from running tests in an isolated, standalone form, meaning that it would be better if it was possible to run specific test of the generated code without the need to run the whole test strip. For this goal, the parameterized way of testing was inefficient, since it entails that every test is generic and there is no way to quickly modify specific parameters in a single test.

So, the test class was re-written to have one method per test, per model. Meaning that instead of ten generic tests ran with parameterized values, there would be fifty [2] specific tests, ten for each test model.

The features tested in Section 5.1.1 were re-tested and their behavior remained, as expected, the same. On top of that, new integration tests were performed to ensure that new features behave as intended, mainly:

- **TC-INT-08 - Business Logic Decoupling:** Verified that when the DSL specifies decoupled business logic, the generated code separates logic into independent modules, allowing tasks to execute without unnecessary dependencies.
- **TC-INT-09 - Executor Behavior:** Configured models with different executor types and confirmed that:
  1. Multi-threaded execution logic is correctly generated when requested.
  2. Static single-thread execution is applied as configured.
- **TC-INT-10 - File Management:** Generated code using models with various file settings and confirmed that:

[1] To add a new test only one function needs to be added, at least.
[2] At that point in time, with five models.

1. Existing files remain unchanged when overwriting is disabled.
2. Configured header prefixes are correctly applied in all files.

- **TC-INT-11 - Extension Script Execution:** Added pre and post generation scripts to the DSL and verified that:
  1. Pre-generation scripts execute successfully before code generation starts.
  2. Post-generation scripts run correctly after generation completes.
- **TC-INT-12 - Build Integration Artifacts:** Configured build integration options in the DSL and confirmed that:
  1. Source files, Makefile, package files, and launch scripts are generated according to the configuration.
  2. Disabling specific build outputs correctly prevents their generation without affecting other features.
- **TC-INT-13 - Legacy Library Handling:** Added legacy library configuration to the DSL and verified that:
  1. Library headers are correctly included in the generated code.
  2. Libraries are copied into the project folder when requested.
- **TC-INT-14 - Target OS Adaptation:** Configured different target operating systems and confirmed that:
  1. Generated code includes OS-specific adaptations for Linux, Windows, and MacOS.

These tests ensured that the new features provided, in fact, additional and specific functionalities that work as predicted.

5.2 Regression Testing

In order to automate the testing of the added features as development progressed unit tests were developed. By comparing the functional version of the generated projects against a newly generated version of the same project, we are able to detect differences in the end result which can point out flaws when a branch of the code generation is modified.

The testing environment remained the same, with only the RAMSES version changing throughout the development.

Essentially, what we test can be visually described by Figure 5.1.

By using this architecture, it is only necessary to verify the correctness of the generated code once. The subsequent generations can be then compared to this already correct premise to determine errors or changes in the code that are not intended.

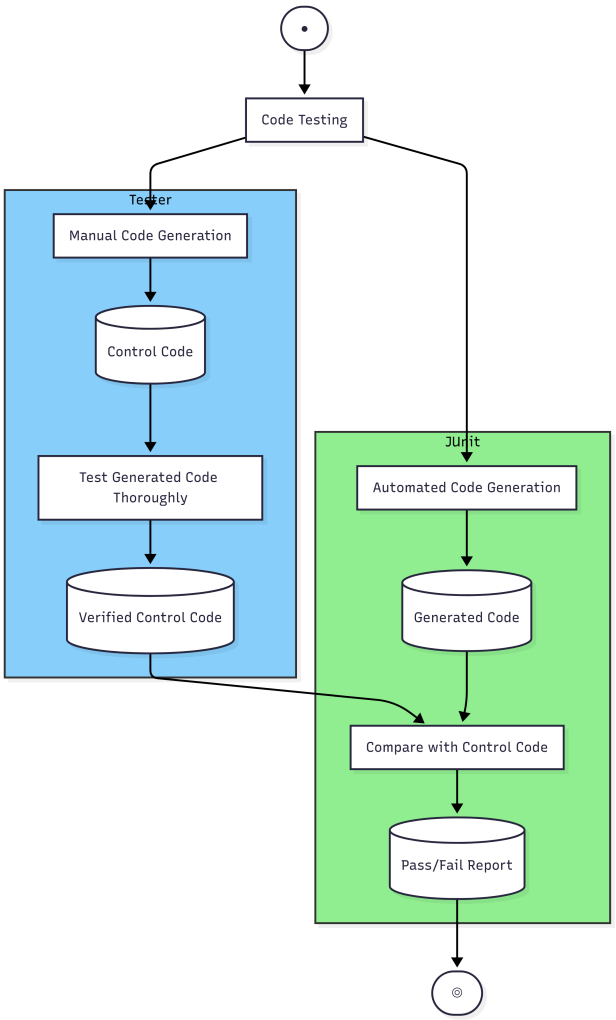

Figure 5.1: Flowchart of the testing strategy for generated code (made by the author)

This verified control code was tested thoroughly by manually generating, compiling, deploying to a target system and executing a series of runtime tests to confirm its functionality.

An obvious flaw of this model is that, not only does it need complete manual validation in the start, but it's also very rigid when it comes to file comparison, comparing each file line-by-line[3].

These flaws can also be seen as benefits, since the code needs to be tested rigorously to ensure maximum code quality when generated. This can only be achieve by actually executing the generated projects and verifying their functionality manually.

The unit tests were made to be as flexible as possible, allowing for different configurations and different models to be tested in a single run, providing insights in multiple generated projects in a single test execution as can be observed in Figure 5.2.

The tests also account for certain projects not generating unneeded files such as the

[3]The comparison is not overly rigid and allows for exceptions such as the generated date and order changes in XML files.

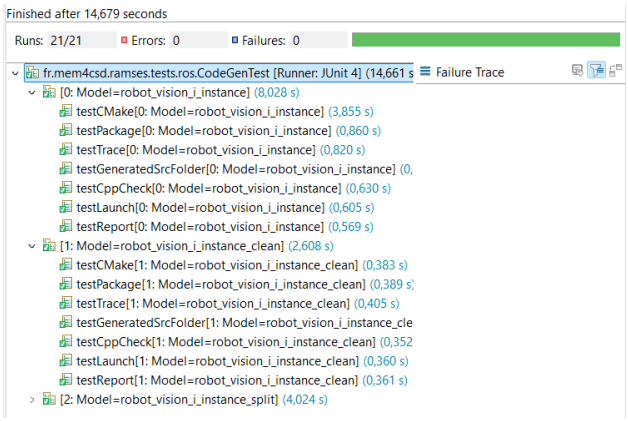

Figure 5.2: Testing results of 3 different models

traceability or the report file. This behavior is translated by simply skipping the test as can be observed in Figure 5.3. Similar tests also verify the existence of folders and files that are a requirement for any ROS project and need to be present after generation.

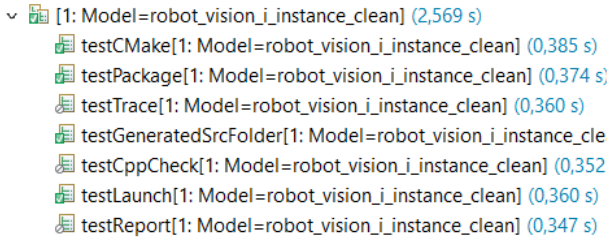

Figure 5.3: Skipped Trace, CppCheck and Report files as those are not present

In addition, the test runs also validate the existence of executable both in the launch and make configurations, notably these executables have to exists in both files.

As a guardrail, if files are present in the control project but not in the generated project that will generate an error and vice versa. This ensures that the generator produces only what is needed and no file is missing.

Table 5.2: Regression test cases and expected outcomes

Test Case ID	Feature	Description	Expected Outcome
TC-REG-01	Regression	Compare output with older version for unchanged models.	Output is functionally identical.
TC-REG-02	Regression	Generate code for previously passing models.	Generation succeeds with no new errors.

TC-REG-03	Regression	Measure generation time for unchanged models.	Timing does not exceed threshold.
TC-REG-04	Regression	Test legacy models without new features enabled.	Output has no formatting or syntax changes.

The tests described in table 5.2 were proven true via JUnit tests, which not only were able to verify the present tests but also additional ones that are not present in this table. The full detailed list of tests can be found in the Appendix A.5.

TC-REG-02, TC-REG-03 and TC-REG-04 were easily proven by the unit test strip since this process can be automated and viewed quickly. TC-REG-01 added a layer of complexity because it pushed for manual verification which had to be done to ensure the functionality of the output code remained the same as the control group.

All the unit tests previously described in this section (Section 5.2) were initially done as plug-in tests, however, due to a multitude of reasons, they were later changed to standalone tests.

- **Plug-in unit tests** are tests done by launching a new Eclipse instance that loads all the resources necessary and executes the unit tests. This approach is simpler to setup but takes significantly more time to run than the standalone option due to the creation of the additional Eclipse instance.
- **Standalone unit tests** do not launch a new Eclipse instance, instead they run detached from eclipse, which makes them faster with the only downside being they require more initial setup to initialize the resources and dependencies that Eclipse would normally provide.

The change from plug-in tests to standalone revealed a clear advantage when it came to test execution time, out of 20 executions there was an average of 20% of time saved, which can be attributed to the standalone runs not having to initialize a new instance of Eclipse. This time reduction is inversely proportional to the amount of tests, as in, less tests will translate to an even faster average of time saved by the standalone execution, which is especially useful when running single or low amounts of tests, having an average of 60% of time saved. Even when running large amounts of tests (>100) there is still a reduced execution time on the standalone run, albeit not as noticeable as before with around 10% reduction. The test run times can be seem in the Appendix A.5.1.

5.3 Usability Validation

With the DSL finalized and its features implemented it was then possible to assess how an end user will interact with the newly developed features. The object of study is the

developed DSL and configuration workflow it uses. The main purpose of this usability validation test is to evaluate, understand, and improve the code configuration DSL and workflow.

This testing will focus particularly on user efficiency (time to complete tasks) and effectiveness (task completion rate) of accomplish tasks, as well as how fast they learn the configuration itself, how often mistakes are made and how comfortable they feel using it. This information will not only be valuable to the current thesis but also to take into account when developing the complete RAMSES tool with an integrated place for the configuration language.

For these tests the participants consisted of end users with moderate AADL knowledge and a highly experienced ROS developer[4] that directly represents the target demographic.

The tests were conducted using the DSL within the RAMSES code generation workflow, operating in a Windows laptop with Osate with the latest versions available at the moment of testing (see more on Section 5.3.5), using it with functional AADL models.

In Goal Question Metric (GQM) terms, this can be translated to:

- **Object**: The developed DSL and its configuration workflow.
- **Purpose**: Evaluate, understand, and improve the workflow.
- **Quality focus**: Assess usability attributes including efficiency, effectiveness, learnability, error rate, and user satisfaction.
- **Perspective**: From the viewpoint of end users with moderate AADL knowledge and a highly experienced ROS developer.
- **Context**: Within the RAMSES code generation workflow, using functional AADL models.

For this test, 5 users with a moderate/high familiarization to AADL were picked to perform a sequence of tasks that mimicked the usage of the DSL to its near fullest extent.

With the goals now in mind, we can also associate research questions to better understand if the goals were actually met in the end, those questions are the following:

- **RQ1 (Effectiveness):** Can users successfully complete the tasks using the DSL without errors?
- **RQ2 (Efficiency):** How quickly can users complete the tasks using the DSL?
- **RQ3 (Learnability):** How quickly do users become proficient with the DSL when performing the sequence of tasks?
- **RQ4 (Error rate):** What types and frequency of errors do users make while using the DSL?

[4] 7+ years of commercial ROS usage.

- **RQ5 (Satisfaction):** How do users perceive the usability of the DSL in terms of comfort, confidence, and overall satisfaction?

Each of the research questions serves to directly evaluate a defined usability attribute (effectiveness, efficiency, learnability, error rate, and satisfaction).

5.3.1 Operational Definition of Usability

According to ISO 9241-11, usability is defined in terms of effectiveness, efficiency, and satisfaction within a specified context of use. Table 5.3 summarizes how each attribute is operationalized and measured in this study.

Table 5.3: Operational definition of usability attributes used in the study

Usability Attribute	ISO 9241-11 Concept	Measurement
Effectiveness	Accuracy and completeness	Percentage of tasks successfully completed without external assistance.
Efficiency	Resources used to achieve goals	Time to complete each task (in seconds). Measured per participant and aggregated across tasks.
Learnability	Time and effort to reach proficiency	Improvement in task completion time and reduction in errors across the ordered sequence of tasks.
Error rate	Frequency and type of user errors	Number and classification of errors observed during each task (syntax misuse, misunderstanding of concepts, navigation errors, etc).
Satisfaction	User comfort and acceptance	Post-task questionnaires including SUS and NASA-TLX ratings (mental demand, effort, frustration, perceived performance).

To ensure the validity of the usability evaluation, it is important to clarify the target population and the sampling strategy used. In the context of the RAMSES tool and its integration with ROS systems, the expected user population consists primarily of:

- Engineers and researchers familiar with AADL with moderate experience in architecture modeling but not necessarily experts in code-generation frameworks or ROS.

- A robotics developer with practical experience in ROS that represented a major demographic expected to interact with the generated code and configuration artifacts.

Given the availability constraints inherent in specialized domains such as AADL and safety-critical modeling, the participant group for this study constitutes a convenience sample. According to Falessi et al. (ST7/ST8) [20], convenience sampling is acceptable in empirical software engineering studies when the objective is exploratory evaluation, early-stage usability validation, and identification of practical issues affecting real users. While this sampling method limits statistical generalizability, it provides valuable insight into typical user behaviour within the actual intended user demographic.

To make this verification operational, each usability requirement introduced in Chapter 3 is mapped to a specific dependent variable in the evaluation design. The requirement that the DSL be human-readable and accessible to users with different technical backgrounds is assessed through learnability and effectiveness metrics, namely the time needed to complete configuration tasks and the rate of successfully completed steps. The requirement for low cognitive overhead and clarity of configuration rules is evaluated using the NASA-TLX workload scale, which measures perceived mental demand, effort and frustration. The requirement that the DSL support correct and error tolerant configuration is linked to the error rate and the qualitative analysis of error types. Similarly, the requirement for a declarative and predictable configuration is associated with efficiency metrics (task duration, need for backtracking) and with participant reports captured through the SUS questionnaire. By associating each requirement with at least one measurable variable, the evaluation ensures that high level design goals such as readability, configurability, robustness, and accessibility are reflected in user experience and performance.

5.3.2 Participants

A total of 5 participants were selected for this study. All were familiar with AADL and had experience with model-driven workflows, even though their backgrounds and technical depth varied. This group reflects the typical user base within the RAMSES ecosystem: developers, testers, and stakeholders who understand AADL semantics but may not necessarily be expert programmers. Participants remained anonymous.

Out of all participants, only one of them interacted with a very early and incomplete version of the DSL and, during the test, this participant did not entice any familiarization with the current DSL, hence its safe to say that no participant understood what the DSL was, what purpose it served and how to operate it before the execution of the test.

5.3.3 Experimental Materials

5.3.4 Tasks

Each participant was asked to perform a sequence of tasks designed to cover every major feature of the configuration language. These tasks simulate realistic use of the DSL and its integration with RAMSES. The task set was organized into five groups:

- **Basic Configuration:** selecting the target language, target operating system, and output folder.
- **Identifier and Naming Options:** changing identifier style, applying prefixes or suffixes, and switching to the *STANDARD* naming convention.
- **Feature Toggles:** enabling or disabling comments, report generation, traceability, and file management options.
- **Advanced Capabilities:** configuring executor type, managing business logic decoupling, referencing legacy libraries, and defining pre and post generation hooks.
- **Full Workflow Validation:** triggering code generation and verifying that the outputs (generated code, reports, trace files, etc) reflect the selected configuration.

The task sequence was intentionally designed to exercise the DSL to near full extent without requiring prior explanation of its internal implementation.

5.3.5 Evaluation Criteria

To assess the usability attributes defined in Section 5.3.1, three types of data were collected during the experimental sessions: quantitative metrics, qualitative observations, and subjective scoring.

The usability evaluation followed the methodology defined in the requirements phase of the project. Both quantitative and qualitative measures were collected to assess the usability of the configuration interface and the effort required to use it.

Quantitative data For every task and participant, the following metrics were recorded:

- Task completion time.
- Number and type of errors committed.
- Number of help requests or clarifications needed.

Qualitative data During the session, the following observations were captured:

- Visible misunderstandings and their severity.
- User comments and verbalized expectations.
- Indicators of the learning curve across tasks (increased speed or confidence).

Subjective evaluation After completing all tasks, each participant completed two standardized questionnaires:

- **SUS** for perceived usability.
- **NASA TLX** to assess perceived workload and cognitive demands.

These instruments provide widely accepted measures for usability and mental effort. They also enable a structured and comparable assessment.

5.3.6 Execution

Each session followed the same procedure:

1. **Introduction:** participants were informed that they would configure a code generator using the provided interface, without receiving task-specific guidance.
2. **Task execution:** participants completed the predefined task sequence while the evaluator recorded times, errors, and observations. No assistance was offered unless explicitly requested.
3. **Post-study questionnaires:** participants completed the SUS and NASA-TLX questionnaires.
4. **Final debriefing:** participants were invited to provide additional comments and suggestions.

All sessions were conducted individually and under identical software environments to ensure comparability.

5.3.7 Analysis

With all data collected, we can proceed with the analysis.

Table 5.4 illustrates the aggregated performance summary of the participants. The completion time is very much acceptable for most tasks given the lack of familiarity with the software that every participant had. We can see a spike when it comes to ROS specific things that the casual AADL user might not be familiar with, for instance, changing the thread executor, managing the business logic, etc.

Table 5.4: Usability Task Performance Summary

Task	Completion Time (s)	Errors	Help Requests
Select target language			
Configure naming style			
Apply identifier prefix/suffix			
Enable/disable comments			
Enable/disable report generation			
Enable/disable traceability			
Change executor type			
Toggle business logic decoupling			
Configure legacy libraries			
Add pre/post scripts			
Trigger code generation			
Verify outputs			
Totals			

The amount of error performed was below expectations, most likely due to the simplicity and structure of the configuration language.

Nevertheless, these errors were quantified in the following Table 5.5.

Table 5.5: Categorization of User Errors

Error Category	Count	Description
Navigation error		Incorrectly locating the configuration page or submenu.
Misinterpretation		Misunderstanding the meaning of an option.
Configuration misuse		Selecting an option that conflicts with the intended task.
UI confusion		Confusion caused by labels, layouts, or grouping.
Missing confirmation		Forgetting to apply/save configuration changes.
Other		

5.3.8 NASA TLX

Dimensions:

- Mental Demand
- Physical Demand
- Temporal Demand
- Performance
- Effort
- Frustration

Each item is scored from 0 to 100.

Table 5.6: NASA-TLX Scores per Participant

Participant	Mental	Physical	Temporal	Performance	Effort	Frustration
P1						
P2						
P3						
P4						
P5						

5.3.9 Discussion

Table 5.7: Qualitative Observations During Testing

Participant	**Observations**
P1	
P2	
P3	
P4	
P5	

5.3.10 Threats to Validity

Although the tests conducted were very promising and revealed a strong structural integrity of the software built, a core weakness of the validation tests was the lack of many participants capable of using ROS in a high degree of proficiency. This was mainly due to the difficulty of having contact and time with a ROS industry individual (our specific target demographic). Nevertheless, our individual testing proved that at least one ROS industry professional approves of the developed configuration language. TODO: verify

Some of the test participants had brief interactions with the DSL throughout the development process which might have slightly influenced their performance even though the numbers do not seem to favor any individual in particular. TODO: verify

5.3.11 Conclusion

Bibliography

[1] Addison-Wesley, 2015 (cit. on p. 25).

[2] In: *Developing Safety-Critical Software* (2017-12), pp. 51–72. DOI: 10.1201/9781315218168-7 (cit. on p. 15).

[3] URL: https://mem4csd.telecom-paristech.fr/blog/index.php/ramses/ (cit. on p. 5).

[4] URL: https://www.mathworks.com/help/simulink/ (cit. on p. 5).

[5] URL: https://www.mathworks.com/help/simulink/automotive-applications.html (cit. on p. 6).

[6] URL: https://wiki.eclipse.org/Acceleo/User_Guide (cit. on pp. 9, 11).

[7] URL: https://www.mathworks.com/help/simulink/ug/create-a-template-from-a-model.html (cit. on p. 11).

[8] URL: https://www.dspace.com/en/pub/home/support/kb/dsutil/kbtladd/tlutil.cfm (cit. on p. 11).

[9] URL: https://eclipse.dev/atl/documentation/old/ATL_Flyer_Normal_Version.pdf (cit. on p. 14).

[10] URL: https://google.github.io/styleguide/cppguide.html (cit. on p. 21).

[11] URL: https://peps.python.org/pep-0008/ (cit. on p. 21).

[12] URL: https://www.eclipse.org/forums/index.php/t/1092427/ (cit. on p. 21).

[13] R. G. Bias and D. J. Mayhew. *Cost-justifying usability*. Academic Press, 1994 (cit. on p. 25).

[14] E. Borde et al. "Architecture models refinement for fine grain timing analysis of embedded systems". In: *2014 25nd IEEE International Symposium on Rapid System Prototyping* (2014-10), pp. 44–50. DOI: 10.1109/rsp.2014.6966691 (cit. on p. 2).

[15] M. Broy et al. "Engineering automotive software". In: *Proceedings of the IEEE* 95.2 (2007-02), pp. 356–373. DOI: 10.1109/jproc.2006.888386 (cit. on p. 1).

[16] M. Consortium. *MISRA C:2025 Guidelines for the use of the C language in critical systems*. The MISRA Consortium Limited, 2025. ISBN: 978-1-911700-19-7 (cit. on pp. 7, 12, 15).

[17] M. Consortium and C. Tapp. *MISRA C++:2023 Guidelines for the use of C++17 in critical systems*. The MISRA Consortium Limited, 2023. ISBN: 978-1911700104 (cit. on pp. 7, 12, 15).

[18] K. Czarnecki and S. Helsen. "Feature-based survey of Model Transformation Approaches". In: *IBM Systems Journal* 45.3 (2006), pp. 621–645. DOI: 10.1147/sj.453.0621 (cit. on p. 2).

[19] R. Debouk. "Overview of the second edition of ISO 26262: Functional Safety—Road Vehicles". In: *Journal of System Safety* 55.1 (2019-03), pp. 13–21. DOI: 10.56094/jss.v55i1.55 (cit. on p. 15).

[20] D. Falessi et al. "Empirical software engineering experts on the use of students and professionals in experiments". In: *Empirical Software Engineering* 23.1 (2017-06), pp. 452–489. DOI: 10.1007/s10664-017-9523-3 (cit. on p. 68).

[21] P. Feiler, B. Lewis, and S. Vestal. "The SAE Architecture Analysis and Design Language (Aadl) a standard for Engineering Performance Critical Systems". In: *2006 IEEE Conference on Computer-Aided Control Systems Design* (2006-10), pp. 1206–1211. DOI: 10.1109/cacsd.2006.285483 (cit. on pp. 1, 4).

[22] R. France and B. Rumpe. "Model-driven development of complex software: A research roadmap". In: *Future of Software Engineering (FOSE '07)* (2007-05), pp. 37–54. DOI: 10.1109/fose.2007.14 (cit. on p. 4).

[23] J. García-García et al. "NDT-Suite: A Methodological Tool Solution in the Model-Driven Engineering Paradigm". In: *Journal of Software Engineering and Applications* 7.4 (2014), pp. 206–217. DOI: 10.4236/jsea.2014.74022 (cit. on p. 4).

[24] O. Khatib and B. Siciliano. *Springer Handbook of Robotics*. Springer International Publishing: Imprint: Springer, 2016 (cit. on p. 1).

[25] E. A. Lee. "Cyber Physical Systems: Design Challenges". In: *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. 2008, pp. 363–369. DOI: 10.1109/ISORC.2008.25 (cit. on p. 1).

[26] K. Mikova. *Why code flexibility is crucial in low-code development?* 2025-02. URL: https://www.appbuilder.dev/blog/code-flexibility (cit. on p. 2).

[27] R. Mittal et al. "Solving the instance model-view update problem in AADL". In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems* (2022-10), pp. 55–65. DOI: 10.1145/3550355.3552396 (cit. on p. 14).

[28] *OpenAPI Generator Configuration*. Accessed: 2025-05-26. URL: https://openapi-generator.tech/docs/configuration/ (cit. on p. 7).

[29] *OpenAPI Generator Plugins*. Accessed: 2025-05-26. URL: https://openapi-generator.tech/docs/plugins/ (cit. on p. 7).

[30] *OpenAPI Generator Template Customization*. Accessed: 2025-05-26. URL: https://openapi-generator.tech/docs/customization/ (cit. on p. 7).

[31] *OpenAPI Generators*. Accessed: 2025-05-26. URL: https://openapi-generator.tech/docs/generators/ (cit. on p. 6).

[32] OpenAPITools. *OpenAPITools/openapi-generator: Openapi generator allows generation of API client libraries (SDK Generation), server stubs, documentation and configuration automatically given an openapi Spec (V2, v3)*. URL: https://github.com/OpenAPITools/openapi-generator (cit. on p. 6).

[33] *OpenModelica - Open Source Modelica-based Modeling and Simulation Environment*. Accessed: 2025-05-27. URL: https://openmodelica.org/ (cit. on p. 6).

[34] *OpenModelica Users Guide: Code Generation and Simulation*. Accessed: 2025-05-27. URL: https://openmodelica.org/doc/OpenModelicaUsersGuide/latest/ (cit. on p. 6).

[35] *OpenModelica Users Guide: Performance Considerations*. Accessed: 2025-05-27. URL: https://openmodelica.org/doc/OpenModelicaUsersGuide/latest/profiler.html (cit. on p. 6).

[36] R. Rajkumar et al. "44.1 Cyber-Physical Systems: The Next Computing Revolution". In: 2010-06, pp. 731–736. DOI: 10.1145/1837274.1837461 (cit. on p. 1).

[37] D. Schmidt. "Guest Editor's Introduction: Model-Driven Engineering". In: *Computer* 39.2 (2006), pp. 25–31. DOI: 10.1109/MC.2006.58 (cit. on pp. 1, 4).

[38] M. Voelter and S. Benz. *DSL Engineering: Designing, implementing and using domain-specific languages*. Dslbook.org, 2013 (cit. on p. 22).

[39] E. Web. *Eclipse EMF*. 2025-03. URL: https://projects.eclipse.org/projects/modeling.emf.emf (cit. on p. 21).

A

Appendix

A.1 Comment Configuration

A.1.1 Comment Configuration Example 1

```
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/int32.hpp"

using namespace std::chrono_literals;

    class UsbCamNd : public rclcpp::Node
    {
        public:

            UsbCamNd(): Node("UsbCamNd")
            {

                image_broadcaster_pub_ = this->create_publisher<std_msgs::msg::Int32>("/usb_cam_nd/image_broadcaster", 10);
                image_broadcaster_timer_ = this->create_wall_timer(500ms, std::bind(&UsbCamNd::image_broadcaster_timer_callback, this));


                usb_spinner_sub_ = this->create_subscription<std_msgs::msg::Int32>("/usb_cam_nd/usbspinner",
                    10, std::bind(&UsbCamNd::usb_spinner_callback, this, std::placeholders::_1));

            }
```

Figure A.1: Example of generated code without comments

When code comments are toggled on, the code from Figure A.1 becomes the code from Figure A.2

Figure A.2 has detailed comments about each component, highlighting and explaining specific code sections.

```cpp

// This node (Usb_Cam_Nd) sets up multiple ROS 2 publishers, subscribers, and timers.
// It uses std_msgs::msg::Int32 as message type for simplicity/demo purposes.
// Each input (subscriber) has a corresponding output (publisher) and timer-based callback.
// Topics follow the pattern: /Usb_Cam_Nd/<component_name>

#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/int32.hpp"

using namespace std::chrono_literals;

    class UsbCamNd : public rclcpp::Node
    {
        public:

            // Constructor: Initializes all components
            UsbCamNd(): Node("UsbCamNd")
            {
                // ---------------- Image_Broadcaster Component ----------------
                image_broadcaster_pub_ = this->create_publisher<std_msgs::msg::Int32>("/usb_cam_nd/image_broadcaster", 10);
                image_broadcaster_timer_ = this->create_wall_timer(500ms, std::bind(&UsbCamNd::image_broadcaster_timer_callback, this));

                // ---------------- UsbSpinner Component ----------------
                usb_spinner_sub_ = this->create_subscription<std_msgs::msg::Int32>("/usb_cam_nd/usbspinner",
                    10, std::bind(&UsbCamNd::usb_spinner_callback, this, std::placeholders::_1));

            }
```

Figure A.2: Example of generated code with comments

A.2 Integration and Regression Tests

Table A.1: Feature dependency table

Feature	Depends On	Notes
Identifiers	-	Core to code structure
Comments	Identifiers	Cheap to implement early
Traceability	Identifiers, Comments	Hard to retrofit later
Report	Traceability	Uses trace data
Standard Compliance	Report, Traceability	Enforce compliance early
Dead Code Elimination	Traceability, Report	Needs stable generation logic
Memory Optimization	Dead Code Elim, Traceability	Impacts data structures directly
Node Interface	Memory Optimization	Enables ROS decoupling
Legacy Code Integration	Node Interface	Most architecture-dependent

A.3 Workflow Execution Time Runs

Table A.2: First run of workflow execution times (in ms) for an average model with the set of configs from A.4

Run	Code	Launch	CMake	Package	Trace	Report	Total Steps	Total	Project Time
1	194	65	54	35	92	208	648	663	348
2	183	88	58	30	85	205	649	674	359
3	138	64	49	34	81	185	551	568	285
4	119	50	38	26	74	159	466	477	233
5	113	51	43	36	76	163	482	495	243
6	104	67	38	27	66	178	480	491	236
7	103	43	34	24	66	147	417	425	204
8	96	45	38	26	77	142	424	438	205
9	89	40	34	24	71	145	403	414	187
10	89	45	35	24	70	154	417	426	193
11	93	42	34	22	65	162	418	425	191
12	86	60	35	25	68	151	425	437	206
13	94	52	37	27	67	153	430	438	210
14	89	41	34	23	62	155	404	413	187
15	87	42	34	24	65	153	405	415	187

When observing Table A.2 we can see that the Trace and Report generation take a more significant amount of time due to their additional checks (line traces and code quality checks respectively).

The downward trend in the first few runs can be attributed to common warmup effects present in development environments. These may include plug-in in initialization, caching and class loading. After these initializations, subsequent runs exhibit more stable and normalized execution times.

A.4 Generator Configuration Options

Complete Project Configuration

Naming Style: `STANDARD`

Affix: `Telecom_`

Affix position: `PREFIX`

Report Generate: `true`

Include traceability: `true`

Include summary tables: `true`

Include ROS summaries: `true`

Show detected errors: `true`

Decoupling Enabled

Comments `true`

Traceability `true`

A.4.1 Configuration UML

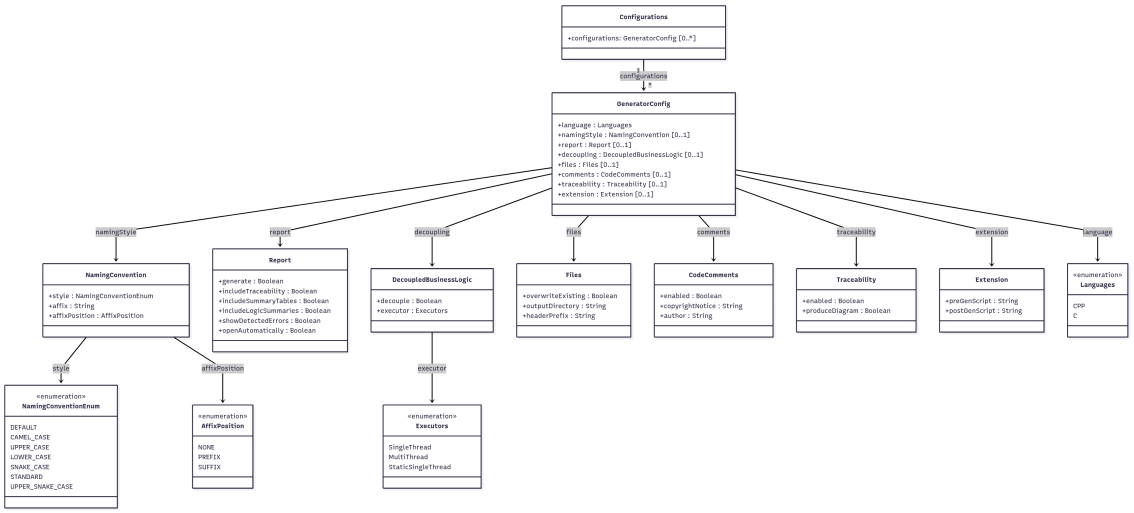

Figure A.3: UML Diagram of the configuration language, made by the author

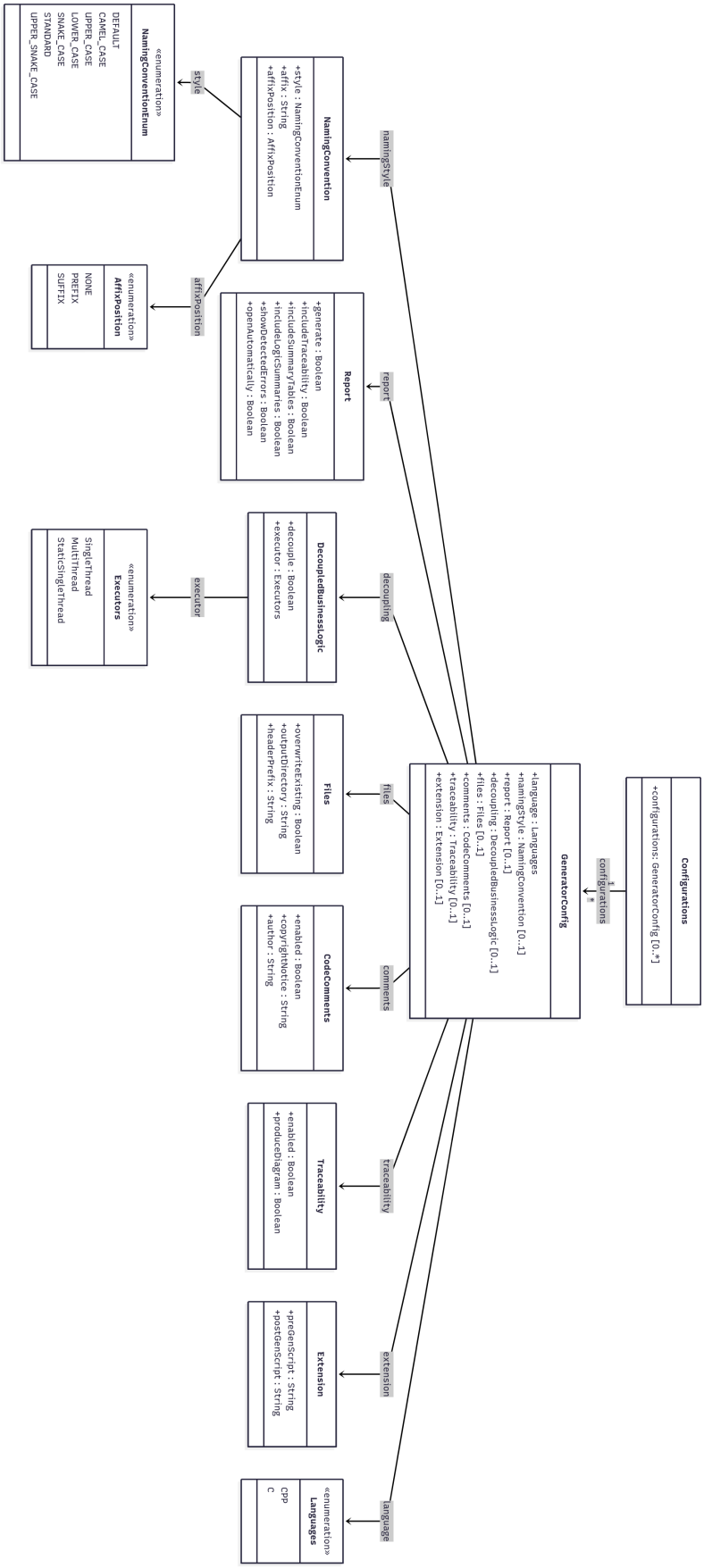

Figure A.4: UML Diagram of the configuration language rotated 90 degrees for better view

A.5 Tests

Table A.3: Code generation verification tests

Test Case ID	Feature	Description
TC-CG-01	ROS Package Structure	Verify presence of expected folders (`src`, `src/launch`) and required files (`CMakeLists.txt`, `package.xml`, `launch.xml`).
TC-CG-02	Source Code Equivalence	Compare generated `src` folder with control reference.
TC-CG-03	Launch File Equivalence	Compare generated `launch.xml` with control reference at node/remap level.
TC-CG-04	CMake Consistency	Compare generated `CMakeLists.txt` with expected reference.
TC-CG-05	Package Metadata	Compare generated `package.xml` with expected reference.
TC-CG-06	Trace Output	Verify `trace.json` against control reference.
TC-CG-07	HTML Report	Compare generated report (`report.html`) with expected.
TC-CG-08	Static Analysis Report	Compare generated `cppcheck-report.xml` with control reference.
TC-CG-09	Node-CMake Consistency	Ensure executables referenced in `launch.xml` are defined as targets in `CMakeLists.txt` (with optional _Node suffix).

TC-CG-10	Colcon Build (Disabled)	Attempt build of generated workspace using `colcon`. Currently disabled due to environment constraints.

The tests observed in Table A.3 were implemented as unit tests. Being extremely useful at keeping feature and runtime consistency during development.

A.5.1 Unit test execution times

Table A.4: Test execution times for standalone and plugin tests

Test Group	Run	Standalone Time (s)	Plugin Time (s)
1 test	1	7.16	16.81
	2	5.58	17.90
	3	5.42	18.06
	4	5.55	16.30
	5	5.42	16.32
10 tests	1	18.59	28.07
	2	18.73	28.50
	3	18.45	28.95
	4	18.60	28.20
	5	18.55	28.10
20 tests	1	27.84	35.97
	2	28.01	36.31
	3	27.90	36.05
	4	27.95	36.10
	5	28.05	36.06
50 tests	1	34.29	41.45
	2	34.27	41.76
	3	34.35	41.50
	4	34.32	41.60
	5	34.30	41.55
>100 tests	1	63.69	71.45
	2	64.91	73.62
	3	64.87	73.26
	4	64.50	72.80
	5	64.75	73.10

Table A.4 highlights how the plug-in and standalone testing environments compare in terms of execution time. The derived conclusion is that standalone testing reduces execution time by a mostly fixed amount (which correlated to the plug-in environment start up). Being beneficial, especially with lower test amounts.

A.5.2 System Usability Scale

The following questions were used in the SUS questionnaire.

1. I would enjoy using this system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need technical support to use this system.
5. I found the various functions in the system were well integrated.
6. I thought there was too much inconsistency in the system.
7. I imagine most people would learn to use this system very quickly.
8. I found the system very cumbersome to use (feels heavy, awkward, slow, or overly complicated).
9. I felt very confident using the system.
10. I needed to learn a lot of things before I could get going with this system.

Responses are recorded on a 5-point Likert scale: 1 = Strongly Disagree, 5 = Strongly Agree.