DEPARTMENT OF
NAME OF THE DEPARTMENT

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

NOVA University Lisbon
Draft: *July 10, 2025*

DEPARTMENT OF
NAME OF THE DEPARTMENT

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

NOVA University Lisbon
Draft: *July 10, 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 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 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 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

Acronyms

AADL Architecture Analysis and Design Language *(pp. 1–7, 12, 13, 16, 17, 19, 20, 23)*
API Application Programming Interface *(pp. 2, 3, 14)*
AUTOSAR AUTomotive Open System ARchitecture *(p. 8)*

CPS Cyber-Physical Systems *(pp. 1, 2)*

DI Department of Computer Science *(p. 2)*
DSL Domain Specific Language *(pp. 3, 15, 17, 21, 23–25)*
DSML Domain Specific Modeling Language *(p. 17)*

IEC International Electrotechnical Commission Standard *(p. 8)*
ISO International Organization for Standardization *(p. 14)*

MBE Model-Based Engineering *(pp. 4, 5)*
MD Model-Driven *(p. 1)*
MISRA Motor Industry Software Reliability Association *(pp. 8, 9, 11, 12, 14)*

RAMSES Refinement of AADL Models for the Synthesis of Embedded Systems *(pp. 1–3, 16, 17, 23)*

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 continues, with continuous increase in system complexity [21], 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 Cyber-Physical Systems (CPS) [20]. 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 [32]. Robotics, as a constituent of CPS, entails unique challenges in software and hardware integration, making problem identification late in the process very expensive.

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 [33]. 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 Architecture Analysis and Design Language (AADL) is a strong modeling language well-suited to embedded systems [17]. 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. [13], 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 one issue: its decisions for generating code 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.

Flexibility to customize code generated is imperative in order to encourage increased adoption by industry and to facilitate integration in diverse development environments [22]. 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 Application Programming Interface (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 Domain Specific Language (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 that already exist, enabling smoother integration of RAMSES into industrial development processes.

Main Contributions:

- A configuration language for RAMSES
- Flexible and customizable C code generation
- Industrial 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 [33]. 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 [19], 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 [18]. 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. 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 is not only strong in its description power but stronger in being capable of supporting early analysis of non-functional properties such as timing, reliability, and safety constraints. 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 (*Refinement of AADL Models for Synthesis of Embedded Systems*) 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.

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.

Yet, despite its advantages, RAMSES is not flawless. Its 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 the only one 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.

Simulink Code Generation For Embedded Systems

Simulink is a flagship Model-Based Design solution, particularly in control systems engineering, developed by MathWorks [3]. 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 [4].

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 [29]. It generates simulation binaries and C code that precisely represent Modelica models and support complex system dynamics and numerical analysis [30].

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 [31]. 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.

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 [28]. Supporting over 40 languages and frameworks [27], 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 [24]. 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 [26, 25].

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

RAMSES vs. Other Code Generators

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

Table 2.1: Code gen configuration feature comparison over multiple tools.

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, 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 such as the utilization of Acceleo, by which RAMSES can be enhanced through the

[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.

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.

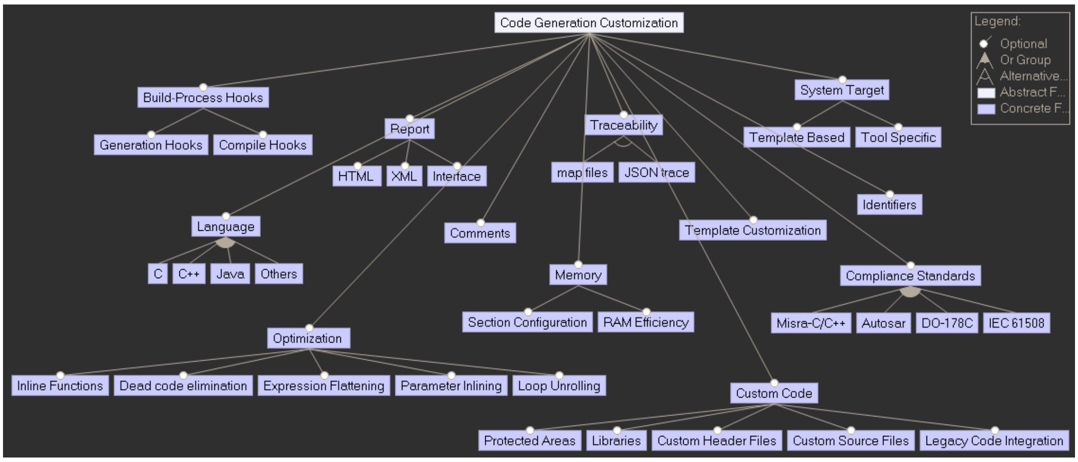

Figure 2.1: Code Gen Config Feature Model

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 contrainsts when dealing with certain features, those constraints are the following:

- 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 model-to-text (M2T) transformation technologies. Among these, Acceleo is highly promising.

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 EMF — Eclipse Modeling Framework format) to text artifacts like source code, documentation, or configuration files [5].

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.

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

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.

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 [5].

- **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 [6].

These approaches allow programmers to mold 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.

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 [7]. 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.

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.

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.

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 [14] and C++ [15] 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 explain 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 an implementation plan.

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, hardcoded in ATL transformations, reflect the assumptions of RAMSES' authors more than the needs of end users [8]. Altering them involves modifying the transformation source itself, often a hard and error-prone task [23].

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. As such, RAMSES provides code generation but not code governance.

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.

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 [14] [15].
- **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 [16].

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.

Integration with Legacy Codebases

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

- Legacy libraries with non-negotiable APIs.
- 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.

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, more on that later.

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, here's a flowchart:

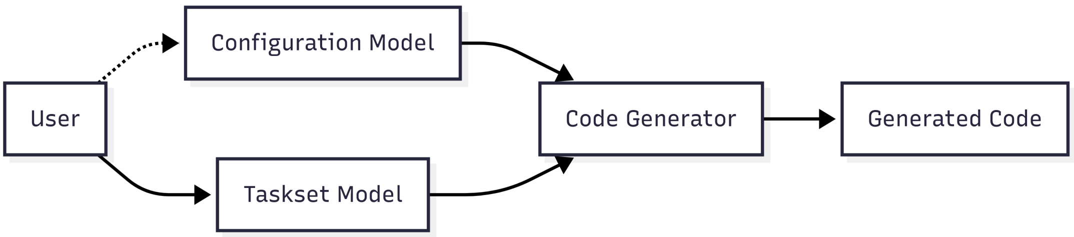

Figure 3.1: Flowchart of the prototype

The user interacts with the main model (Taskset in this case) and *can* interact with the configuration model aswell, if thats 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 therms 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, and, since the placeholder code has already been defined and altered to allow for dinamic 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:

Taking in the result from Figure 3.3 we can get an idea of the default, unaltered code generation, but 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.

```
void T1();

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

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

```
void t_1();

// Initialization of data for task T1 (periodic, period = 1000 ms)
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 behaviour 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 multiple users that were familiar 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, many users defined it 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, with this functionality, appears the ability to create multiple versions of the functionally same code just with naming differences. This also aligns with the coding standards for multiple languages, given that some languages require identifiers to use a certain nomenclature [9, 10], which can clash with AADL node names.

[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

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 to 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, as we want to modify the 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 lines. [35] As the configuration model is built on Ecore, the addition of new naming styles is not possible directly by the user, this means two things: in order to enter a wrong coding style the user needs to have high technical knowledge of the system[6]. If instead we wanted to **add** a new naming style, that is very much possible with a quick modification of the Configurator's Ecore model and the Acceleo naming format algorithm.

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 [11]. 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.

[6]This can change with the proper DSL implementation, however, configuration checks will be applied then, removing this problem.

3.7 System Evaluation

In order to assure that the DSL meets expectations [34], 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].

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.

DSL existence

This requirement will be fairly straightforward to prove since it implies a direct usage of the DSL in order 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.

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.

Usability

The configuration language should be easily usable, even by end users with limited programming experience, in order 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, allowing the comparison of usability and learning curve.

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

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

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

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.

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 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.

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.

3.8 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 [12]. 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.

- **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

Development Plan

This chapter details the development plan for the configuration language, its implementation, testing, documentation, etc

The implementation of the proposed DSL requires a methodical and objective approach in terms of timing and task management. For that reason, a Gantt chart was made with the intent of guiding the development and ensuring the end product meets expectations.

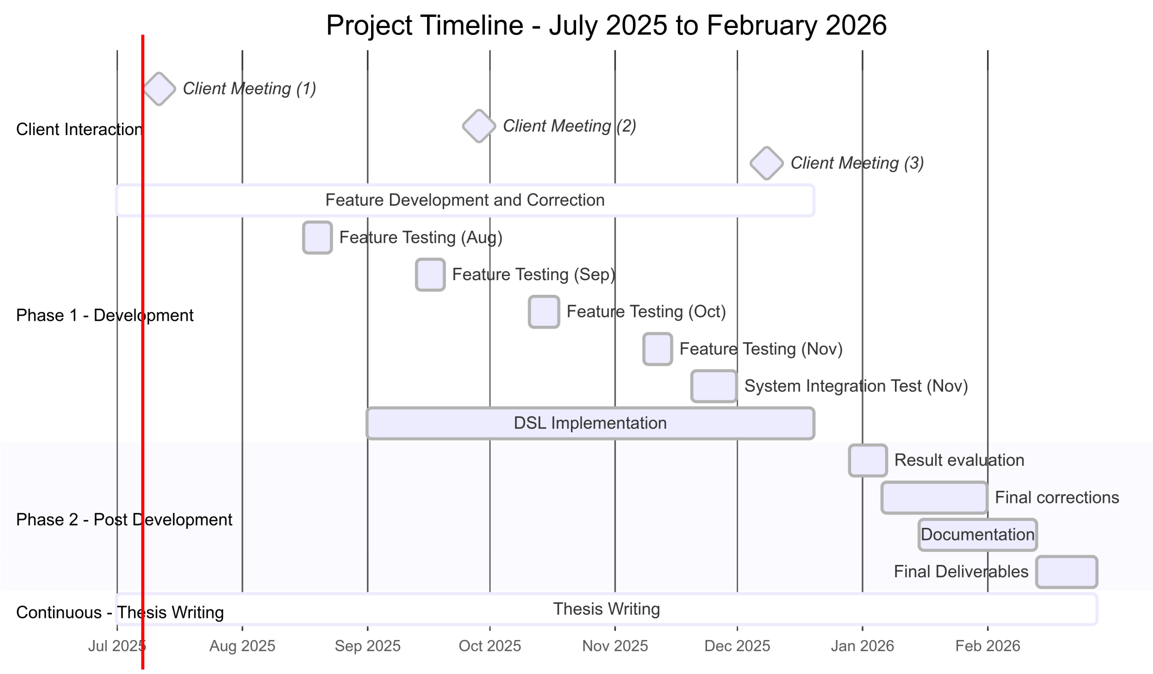

Figure 4.1: Thesis planning Gantt chart

As we can see in Figure 4.1, the plan is separate in two distinct phases: Development and Documentation. Those phases overlap with both client meetings and the writing of the thesis, which is done in parallel with other tasks from start to finish.

4.1 Client Interaction

Client meetings are present to ensure that the features developed are useful and needed addition to the end user. The first and second meeting are mostly interviews directed to acquire quick feedback on the matter, while the third meeting will focus on the evaluation and testing of the mostly finished code generator configurator.

4.2 Development Phase

The development follows an iterative nature, which means that once a feature is developed, it is tested, documented and corrected when needed. Each month, full system wide testing, is also conducted to ensure multiple features work in harmony with each other. A system integration test is also performed near the end of the development phase to ensure that end users can use the system effectively and can achieve goals unachievable before. This is a critical point in the development since it will be used in addition to the third client meeting to build a system evaluation report.

These will be the types of testing used and their goals:

- **Unit Testing** during development, as new features get developed. Ensures new features meet their goals and catches early bugs.
- **Regression Testing** done monthly to make sure previous features are still working with the addition of new ones.
- **Integration Testing** done in addition to Regression Testing to test how different parts of the system work together.
- **Validation Testing** after development to validate that the features developed are in line with the requests of the end users and clients that will be testing the whole system.

Parallel to the code generator configuration feature development, a DSL will also be developed. Near the end of the development of a specific feature we need to also implement its actual configuration functionality via the DSL which will provide a way for end users to use the configurator, hence its parallel development method which also includes testing. Both the feature and DSL development go beyond the last evaluation and testing in order to ensure a safety gap for last minute features or corrections.

4.3 Documentation Phase

The final phase begins with the analysis of previously gathered evaluation results and insights on end user opinion, thought process and interaction when using the configuration language. Its a very important step since it will allow for quick overview on implementation problems, bugs and other suggestions that might have not been thought of before. After that comes the documentation part for both users and future developers, since the configuration language will be very extensible. Documentation, which includes user manuals, developer guides and system documentation, will also be finished during development, however this interval mainly serves to make sure everything is documented properly.

The final part consists in ensuring every deliverable is correct and ready to send. It mostly include meetings and information exchange with every involved party to validate the work done, while not directed to development or corrections, it provides another safe window to make changes if strictly needed.

4.4 Final Considerations

The overall structure of the Gantt chart is clear and direct, although it provides a detailed and structured timeline, in practice it will serve mostly as a base and the development might end up with unforeseen requirements depending on the necessities of the parts involved, or technical challenges. The chart provides clarity and direction but remains adaptable to eventual changing project needs.

Bibliography

[1] Addison-Wesley, 2015 (cit. on p. 23).

[2] In: *Developing Safety-Critical Software* (2017-12), pp. 51–72. DOI: 10.1201/9781315218168-7 (cit. on p. 14).

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

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

[5] URL: https://wiki.eclipse.org/Acceleo/User_Guide (cit. on pp. 9, 10).

[6] URL: https://www.mathworks.com/help/simulink/ug/create-a-template-from-a-model.html (cit. on p. 10).

[7] URL: https://www.dspace.com/en/pub/home/support/kb/dsutil/kbtladd/tlutil.cfm (cit. on p. 10).

[8] URL: https://eclipse.dev/atl/documentation/old/ATL_Flyer_Normal_Version.pdf (cit. on p. 13).

[9] URL: https://google.github.io/styleguide/cppguide.html (cit. on p. 19).

[10] URL: https://peps.python.org/pep-0008/ (cit. on p. 19).

[11] URL: https://www.eclipse.org/forums/index.php/t/1092427/ (cit. on p. 20).

[12] R. G. Bias and D. J. Mayhew. *Cost-justifying usability*. Academic Press, 1994 (cit. on p. 23).

[13] 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).

[14] 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. 11, 14).

[15] 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. 11, 14).

[16] 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. 14).

[17] 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 p. 1).

[18] 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).

[19] 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).

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

[21] 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).

[22] 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).

[23] 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. 13).

[24] *OpenAPI Generator Configuration*. Accessed: 2025-05-26. URL: https://openapi-generator.tech/docs/configuration/ (cit. on p. 6).

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

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

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

[28] 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).

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

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

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

[32] 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).

[33] 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).

[34] M. Voelter and S. Benz. *DSL Engineering: Designing, implementing and using domain-specific languages*. Dslbook.org, 2013 (cit. on p. 21).

[35] E. Web. *Eclipse EMF*. 2025-03. URL: https://projects.eclipse.org/projects/modeling.emf.emf (cit. on p. 20).