NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

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.

SOME THOUGHTS ON THE LIFE, THE UNIVERSE, AND EVERYTHING ELSE

Dissertation Plan
MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
Draft: *June 26, 2025*

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
NAME OF THE DEPARTMENT

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

SOME THOUGHTS ON THE LIFE, THE UNIVERSE, AND EVERYTHING ELSE

AURÉLIO MIRANDA SANTOS RODRIGUES GABOLEIRO

BSc in Computer Science and Engineering

Adviser: Mary Doe Adviser Name
Full Professor, NOVA University Lisbon

Co-advisers: John Doe Co-Adviser Name
Associate Professor, NOVA University Lisbon

John Doe other Co-Adviser Name
Full Professor, NOVA University Lisbon

Dissertation Plan
MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
Draft: *June 26, 2025*

Abstract

The automatic generation of code from templates is an approach widely adopted in the industry to reduce costs and increase software reliability. However, this generation has to be highly configurable to meet specific requirements, such as good coding practices, compatibility with legacy 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 adaptable configuration of the generated code.

This project proposes the design and implementation of a configuration language for RAMSES to allow code generation to be customized according to the requirements of each industry. The steps consist of defining the syntax and semantics of the language, integrating the language 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 useful 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 adotada na indústria para reduzir custos e aumentar a fiabilidade do software. No entanto, esta geração tem de ser altamente configurável para satisfazer requisitos específicos, tais como boas práticas de codificação, compatibilidade com APIs antigas 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 código automaticamente a partir de modelos de alto nível, suprimindo 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 se adaptarem a ambientes industriais específicos requer uma configuração adaptável do código gerado.

Este projeto propõe a conceção e a implementação de uma linguagem de configuração para o RAMSES, de modo a permitir que a geração de código seja personalizada de acordo com os requisitos de cada indústria. As etapas consistem em definir a sintaxe e a semântica da linguagem, integrar a linguagem no RAMSES e testar a mesma 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 bonnes pratiques de codage, la compatibilité avec les API existantes et l'optimisation des performances.

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 significatif 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 afin de permettre la personnalisation de la génération de code en fonction des exigences de chaque industrie. Les étapes consistent à définir la syntaxe et la sémantique du langage, à intégrer le langage dans RAMSES et à le tester à travers des 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 produire 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

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 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 [17], 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) [16]. 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 [26]. 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 [27]. 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 [11]. 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. [4], the Refinement of AADL Models for the Synthesis of Embedded Systems (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 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 [18]. 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 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:** Describes the architecture and design of the configuration language proposed for RAMSES.
- **Chapter 4:** Presents the integration and application of the configuration mechanism within the RAMSES toolchain.
- **Chapter 5:** Ensures the efficacy of the given approach with experimental case studies and tests.

- **Chapter 6:** Ends the thesis by providing an overview of contributions and possible areas of future research.

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. 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 [13], 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. 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 **Architecture Analysis & Design Language (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 a focus on code generation 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.

This constraint will be a discussed throughout this thesis. In subsequent chapters, we will return to RAMSES to discuss its architecture in greater depth and look at potential ways to make it more configurable. ((Should i discuss architecture here actually? -A))

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. 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.

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 [23]. It generates simulation binaries and C code that precisely represent Modelica models and support complex system dynamics and numerical analysis [24].

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 [25]. 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. Supporting over 40 languages and frameworks [22], it is widely used across software projects.

Generation is controlled via configuration files (JSON or YAML) that specify package naming, class prefixes, data type mappings, and code style, enabling consistent architectural and coding standards [19]. 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 [21, 20].

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.

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

Feature	Simulink	OpenModelica	OpenAPI	RAMSES
Identifiers[1]	Yes	No	Yes	No
Optimization	Yes	No	No	No
Legacy Code Integration	Yes	No	Yes[2]	No

As can be observed, Simulink—a commercial, high-end software—thoroughly 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.

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 introduction of greater configurability and extensibility, and thereby narrowing the gap with more mature tools in the domain.

[1] Names of Functions, Classes, Variables, etc
[2] User-driven process

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 a highly promising candidate.

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.

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.
- **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.

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. 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

Finally, 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 (Motor Industry Software Reliability Association) C and C++ 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.

3

Challenges and Requirements for a Configurable Code Generator

The ability to generate code is no longer sufficient: the code must be understandable, auditable, integrable, and compliant with a spectrum of project-specific requirements. This chapter delves into why configurability is no longer optional, and how traditional code generation pipelines must evolve.

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 (e.g., 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 downstream consumers. Altering them involves modifying the transformation source itself — often a non-trivial and error-prone task.

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 out-of-the-box way to enforce such 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 [5] [6].
- **DO-178C:** Requires traceability, tool qualification, and clear derivation from high-level requirements.
- **ISO 26262:** Enforces safety-related development practices and documentation.

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'll 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 such a language is non-trivial. It must strike a delicate balance between expressiveness, usability, and integration. Based on industrial feedback and analysis of RAMSES' shortcomings, the following characteristics are proposed:

1. Declarative, Not Imperative

Users should describe *what* they want (e.g., "all functions must use snake_case") rather than *how* to achieve it. This aligns with the model-driven philosophy and supports better static analysis.

2. Human-Readable and Tool-Accessible

The configuration format (YAML, JSON, or a DSL) should be version-control friendly, diffable, and understandable to engineers. It must also be machine-readable for validation and generation.

3. Modular and Hierarchical

Configurations should support reuse and composition. For example, a company-wide 'default_config' can be extended by a project-specific override. Inheritance models must be explicit and predictable.

4. Aligned with Modeling Concepts

To maintain semantic coherence, configuration keys should refer to modeling elements (e.g., 'thread', 'port', 'data component') and not generator internals.

5. 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.

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 (e.g., RTOS tasks, processes, memory partitions).
3. **Memory Allocation Policies:** Whether to use static, stack, or dynamic memory for component instances or data buffers.
4. **Code Instrumentation:** Hooks for logging, tracing, or runtime checks (e.g., insertion of 'assert' or instrumentation macros).
5. **Conditional Feature Flags:** Ability to enable or disable parts of the generator (e.g., generate test stubs, insert HAL wrappers).
6. **Toolchain-Specific Directives:** Integration of compiler pragmas, section attributes, or OS-specific macros.

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, memory behavior, traceability, and lifecycle.

3.6 Conclusion: Toward a Generator for System Integrators

RAMSES, in its current form, reflects a well-engineered proof-of-concept suitable for academic settings or highly controlled environments. However, modern embedded systems

development — especially in regulated industries — demands more than correctness. It demands adaptability, traceability, and maintainability.

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

The next chapter will formalize this idea by designing a configuration language tailored to RAMSES. The goal: decouple transformation logic from project policy and pave the way for truly industrial-scale model-driven code generation.

4

Adding Support to a New School (work in progress)

My advice to customize the novathesis template to another School/University/Department/Degree is to browse the existing supported degrees to find one that is *close enough*, and depart from there!

The multitude of layouts supported by the novathesis template is based in a three-tier naming scheme, separated by slashes: University / School / Department-or-Degree. This three-tier naming scheme is also reflected in a three-tier directory (folder) structure in: `<project_root>a/NOVAthesiFiles/Schools/...`. For example:

```
...
|
+-- nova
|   +-- Images
|   +-- fct
|   |   \-- Images
|   +-- ims
|   |   \-- Images
|   ...
|
\-- uminho
    +-- Images
    +-- ea
    |   \-- Images
    +-- ec
    |   \-- Images
    ...
```

The directory `uminho` contains the customization for all Schools of Universidade do Minho. This university is an example of the case where the regulations are defined at

University level and all the schools apply the same thesis layout and organization. So, the all the customization is done in the file `uminho/uminho-defaults.ldf`, except the definition of the name and logo of each individual school.

As another example, the directory `nova` contains the customization for all Schools from NOVA University Lisbon. This university grants a lot of freedom in the definition of the thesis layouts. In some cases, they are defined at the School level (e.g., NOVA FCT), while is some other cases they are defined separately for each degree (e.g., NOVA IMS).

1. Try all the already supported schools and check which one is closer to your needs;
   a) Edit `Config/1_novathesis.tex` and near line 28 uncomment the line with key `\ntsetup{school=<SOMETHING>}`;
   b) For each school supported (see the comment), replace `<SOMETHING>` with the school name, e.g., make it `\ntsetup{school=ulisboa/fmv}`
   c) Recompile and check the document. Particularly, check the cover layout, the front-page (second cover) layout, the front-matter contents, the bibliography style;
   d) Repeat for the next school, until you find one close enough.
2.

Bibliography

[1] C. Artho, K. Havelund, and A. Biere. *High-Level Data Races*. 2003. URL: citeseer.ist.psu.edu/artho03highlevel.html (cit. on p. 27).

[2] C. Artho, K. Havelund, and A. Biere. "Using Block-Local Atomicity to Detect Stale-Value Concurrency Errors". In: *ATVA*. Ed. by F. Wang. Vol. 3299. Lecture Notes in Computer Science. Springer, 2004, pp. 150–164. ISBN: 3-540-23610-4 (cit. on pp. 27, 28).

[3] N. E. Beckman, K. Bierhoff, and J. Aldrich. "Verifying Correct Usage of Atomic Blocks and Typestate". In: *SIGPLAN Not.* 43.10 (2008), pp. 227–244. ISSN: 0362-1340. DOI: http://doi.acm.org/10.1145/1449955.1449783 (cit. on pp. 27, 28).

[4] 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).

[5] 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 p. 13).

[6] 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 p. 13).

[7] W. contributors. *Shunting-yard algorithm — Wikipedia, The Free Encyclopedia*. 2017-03. URL: https://en.wikipedia.org/w/index.php?title=Shunting-yard_algorithm&oldid=817901155 (visited on 2018-03-01) (cit. on p. 24).

[8] R. J. Dias et al. "Verification of Snapshot Isolation in Transactional Memory Java Programs". In: *Proceedings of the 26th European conference on Object-oriented programming (ECOOP'12)*. Springer-Verlag, 2012-06 (cit. on p. 23).

[9] J. Eliot and B. Moss. *Nested transactions: an approach to reliable distributed computing*. Cambridge, MA, USA: Massachusetts Institute of Technology, 1985. ISBN: 0-262-13200-1 (cit. on p. 27).

[10] *Fast Lexical Analyser*. URL: https://github.com/westes/flex (visited on 2020-07-26) (cit. on p. 24).

[11] P. Feiler, B. Lewis, and S. Vestal. "The SAE Architecture Analysis AMP; 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. 2).

[12] C. Flanagan and S. N. Freund. "Atomizer: a dynamic atomicity checker for multithreaded programs". In: *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Venice, Italy: ACM, 2004, pp. 256–267. ISBN: 1-58113-729-X. DOI: http://doi.acm.org/10.1145/964001.964023 (cit. on p. 28).

[13] 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. 5).

[14] *Gnu Bison*. URL: https://www.gnu.org/software/bison/ (visited on 2020-07-26) (cit. on p. 24).

[15] *IBM's Concurrency Testing Repository* (cit. on p. 28).

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

[17] 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).

[18] 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).

[19] *OpenAPI Generator Configuration*. Accessed: 2025-05-26. URL: https://openapi-generator.tech/docs/configuration/ (cit. on p. 8).

[20] *OpenAPI Generator Plugins*. Accessed: 2025-05-26. URL: https://openapi-generator.tech/docs/plugins/ (cit. on p. 8).

[21] *OpenAPI Generator Template Customization*. Accessed: 2025-05-26. URL: https://openapi-generator.tech/docs/customization/ (cit. on p. 8).

[22] *OpenAPI Generators*. Accessed: 2025-05-26. URL: https://openapi-generator.tech/docs/generators/ (cit. on p. 7).

[23] *OpenModelica - Open Source Modelica-based Modeling and Simulation Environment*. Accessed: 2025-05-27. URL: https://openmodelica.org/ (cit. on p. 7).

[24] *OpenModelica Users Guide: Code Generation and Simulation*. Accessed: 2025-05-27. URL: https://openmodelica.org/doc/OpenModelicaUsersGuide/latest/ (cit. on p. 7).

[25] *OpenModelica Users Guide: Performance Considerations*. Accessed: 2025-05-27. URL: https://openmodelica.org/doc/OpenModelicaUsersGuide/latest/profiler.html (cit. on p. 7).

[26] 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).

[27] 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 p. 1).

[28] N. Shavit and D. Touitou. "Software transactional memory". In: *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*. Ottowa, Ontario, Canada: ACM, 1995, pp. 204–213. ISBN: 0-89791-710-3. DOI: http://doi.acm.org/10.1145/224964.224987 (cit. on p. 27).

[29] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. Fifth. McGraw-Hill, 2006. ISBN: 007-124476-X (cit. on p. 27).

[30] C. von Praun and T. R. Gross. "Static Detection of Atomicity Violations in Object-Oriented Programs". In: *Journal of Object Technology*. 2003, p. 2004 (cit. on p. 28).

A

NOVAthesis covers showcase

This Appendix shows examples of covers for some of the supported Schools. When the Schools have very similar covers (e.g., all the schools from Universidade do Minho), just one cover is shown. If the covers for MSc dissertations and PhD thesis are considerable different (e.g., for FCT-NOVA and UMinho), then both are shown.

temp

A.1 A section here

temp

B

Appendix 2 Lorem Ipsum

This is a test with citing something [8] in the appendix.

C

A Short LaTeX Tutorial with Examples, i will delete this late on

This Chapter aims at exemplifying how to do common stuff with LaTeX. We also show some stuff which is not that common! ;)

Please, use these examples as a starting point, but you should always consider using the *Big Oracle* (aka, Google, your best friend) to search for additional information or al-ternative ways for achieving similar results.

C.1 Document Structure

C.2 Dealing with Bibliography

Citing something online [7, 10, 14].

C.3 Inserting Tables

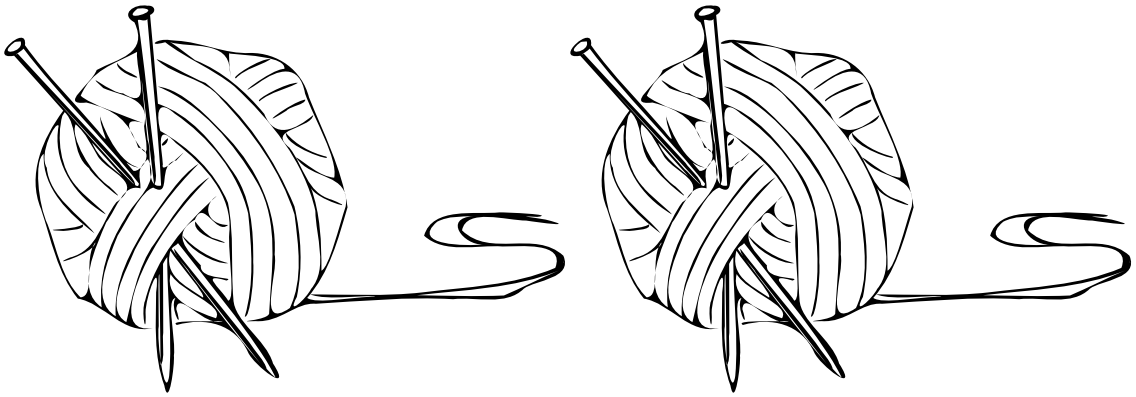
C.4 Importing Images

C.5 Floats, Figures and Captions

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.



(a) One sub-figure (b) Another sub-figure

Figure C.1: A figure with two sub-figures!

And this is a small text that references the Figure C.1 and its Subfigures C.1(a) and C.1(b).

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan

eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

C.6 Text Formatting

C.7 Generating PDFs from LATEX

C.7.1 Generating PDFs with pdflatex

You may create PDF files either by using `latex` to generate a DVI file, and then use one of the many DVI-2-PDF converters, such as `dvipdfm`.

Alternatively, you may use `pdflatex`, which will immediately generate a PDF with no intermediate DVI or PS files. In some systems, such as Apple, PDF is already the default format for LATEX. I strongly recommend you to use this approach, unless you have a very good argument to go for `latex` + `dvipdfm`.

A typical pass for a document with figures, cross-references and a bibliography would be:

```
$ pdflatex template
$ bibtex template
$ pdflatex template
$ pdflatex template
```

You will notice that there is a new PDF file in the working directory called `template.pdf`. Simple :)

Please note that, to be sure all table of contents, cross-references and bibliographic citations are up-to-date, you must run `latex` once, then `bibtex`, and then `latex` twice.

C.7.2 Dealing with Images

You may process the same source files with both `latex` or `pdflatex`. But, if your text include images, you must be careful. `latex` and `pdflatex` accept images in different (exclusive) formats. For `latex` you may use EPS ou PS figures. For `pdflatex` you may use JPG, PNG or PDF figures. I strongly recommend you to use PDF figures in vectorial format (do not use bitmap images unless you have no other choice).

C.7.3 Dealing with Citations

Para fazer citações, deverá usar-se a chave da referência no ficheiro BibTeX. Se for uma única referência [2], usar um "~" para ligar o `\cite{...}` à palavra que o precede (... `referência~\cite{Artho04}`). Caso queira fazer múltiplas citações [28, 29, 9], deverá agrupá-las dentro de um úinico `\cite{...}`.

Note que o ficheiro de bibliografia pode ter tantas entradas quantas quiser. Apenas aquelas cuja chave seja referenciada no texto é que serão incluidas na listagem de bibliografia.

C.7.4 Footnotes

Footnotes[1] will be numbered and shown in the bottom of the page.

C.7.5 Tables

The Table C.1 illustrates some important concepts associated with table construction:

i) Do not use vertical lines;

ii) The caption should be above the table;

iii) Use the macros `\toprule`, `\midrule` and `\bottomrule` to make the top, inner and bottom horizontal lines, respectively.

Table C.1: Test results summary.

Test	Anomalies	Warnings	Correct	Categories	Missed
Connection [3]	2	2	1	*C*	1
Coordinates'03 [1]	1	4	1	*2B*, *1C*	0
Local Variable [1]	1	2	1	*A*	0
NASA [1]	1	1	1	—	0
Coordinates'04 [2]	1	4	1	*3C*	0
Buffer [2]	0	7	0	*2A*, *1B*, *2C*, *2D*	0

[1] This is a simple footnote.

Double-Check [2]	0	2	0	*1A, 1B*	0
StringBuffer [12]	1	0	0	—	1
Account [30]	1	1	1	—	0
Jigsaw [30]	1	2	1	*C*	0
Over-reporting [30]	0	2	0	*1A, 1C*	0
Under-reporting [30]	1	1	1	—	0
Allocate Vector [15]	1	2	1	*C*	0
Knight Moves [3]	1	3	1	*2B*	0
Total	12	33	10	5A, 6B, 10C, 2D	2

C.7.6 Figures

The images inserted in the document must be of good quality, preferably in vector format (vector PDF) and not in *bitmap* (PNG, JPG, etc.). *bitmap* images (Figure C.2) do not scale well and have negative effects on the quality of your document. On the other hand, *vector* images Figure C.3 scale as much as necessary without degrading the quality of the image.

You should only use *screenshots* for your plots, charts, etc, if you absolutely have no other alternative. Instead of generating a *screenshot*, try using a virtual PDF printer and printing to a PDF file. As a general rule, you will get a vector PDF. Even if your PDF contains images, they will always be of higher or equal quality than what you would get with a *screenshot*.

To combine several figures into a single one... You can then reference the set as Figure C.4 or the sub-figures separately as C.7.6 and C.7.6.

C.8 Equations

LaTeX is a powerful tool for writing in a mathematical style. It allows you to insert formulas into the text, such as this: $ax^2 + bx + c = 0$. It also allows formulas to be highlighted on a separate line and centered on the page.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

or numbererd

$$e = mc^2 \tag{C.1}$$

which can latter be referenced as equation C.1

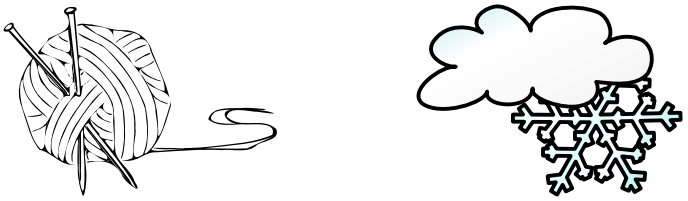
C.9 Test for algorithms

Uncomment the algorithms source below and add the following to file "`5_packages.tex`"

```
\usepackage{algorithm2e}
```

Figure C.2: Bitmap image (JPG/PNG)

Figure C.3: Vectorial image (PDF)



(a) Novelo de lã (b) Tempestade com neve

Figure C.4: Exemplo de utilização de *subbottom*

```
\RestyleAlgo{ruled}
```

and uncomment

```
\ntaddlistof{listofalgorithms}
```

in file "`8_list_og.tex`".

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

I

Annex 1 Lorem Ipsum