Technische Universität München
TUM School of Computation, Information and Technology

Automatic Generator Methodology for Safe Embedded Software

Michael Werner

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

genehmigten Dissertation.

Vorsitz:	Prof. Dr.-Ing. Wolfgang Kellerer
Prüfer der Dissertation:	1. Hon.-Prof. Dr.-Ing. Wolfgang Ecker
	2. Priv.-Doz. Dr.-Ing. habil. Daniel Müller-Gritschneder

Die Dissertation wurde am 23.03.2023 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 24.11.2023 angenommen.

"The function of good software is to make the complex appear to be simple."

(Grady Booch, Founder of the Unified Modeling Language)

Abstract

The growing complexity of embedded systems requires new strategies to meet the challenges of embedded system development. The evolution of programming languages shows that abstraction is the best way to overcome complexity. Model-driven engineering is a methodology that raises the level of abstraction and attempts to approximate human language, thinking, and conceptualization. Combined with a code generation approach, the development process shifts from manual coding to generator-based development. Instead of manually implementing hardware (e.g., VHDL, Verilog) or software code (e.g., C, `C++`), generators coded in high-level languages generate these design views.

While such modern development techniques are becoming more and more accepted in hardware development, embedded software is still predominantly implemented manually. Consequently, software development becomes even more of a bottleneck than it already is, undoing the improvements made in HW development. This thesis addresses this shortcoming and proposes an embedded software design methodology based on model-driven architecture to reduce costs, increase quality and speed-up embedded software development. Together with the existing hardware modeling approach, a holistic generation flow is created that ensures the synchronized generation of hardware and embedded software.

The flow proposes an embedded software metamodel to abstract the structure and behavior of embedded software. Through the holistic approach, transformations translate a specification (conceptual model) into instances of the embedded software metamodel and the hardware metamodel (hardware generation flow). They result in a ready-to-use embedded software code whose structure and behavior are tailored to the particular hardware device. The presented approach handles the entire embedded software stack. For the creation of the HW/SW interface, an automatism is set up to optimize the memory layout of the peripherals in order to implement an efficient hardware abstraction layer. The approach follows several transformation steps for the device driver layer to create a wide variety of device driver variants. A user can further customize the variant according to the application and system needs. The generation flow supports (1) highly configurable device specifications, (2) multiple embedded software architecture schemes, (3) non-functional extensions (e.g., safety measures), and (4) different platforms (e.g., languages).

Applying this approach in the industry has high potential. A developer must no longer be a domain expert to implement, e.g., safe embedded software. This saves a lot of development cost while being less error-prone as it replaces repetitive implementation activities. Furthermore, the designer can analyze several design alternatives to identify the best variant for a specific system.

Zusammenfassung

Die wachsende Komplexität eingebetteter Systeme erfordert neue Strategien um Komplexität bei der Entwicklung von eingebetteten Systemen zu bewältigen.Die Evolution der Programmiersprachen zeigt, dass Abstraktion der beste Weg zur Bewältigung von Komplexität ist. Modellbasierte Entwicklungsmethoden heben das Abstraktionslevel an und versuchen, sich der menschlichen Sprache, dem Denken und der Konzeptualisierung anzunähern. Zusammen mit der Codegenerierung verlagert sich manuelle Codierung hin zur generatorgestützten Entwicklung. Statt Hardware- (z.B. VHDL, Verilog) oder Software-Code (z.B. C, `C++`) manuell zu implementieren, können Generatoren, die in Hochsprachen kodiert sind, diese Entwurfssichten erzeugen.

Während sich in der Hardware-Entwicklung zunehmend solche neuen Entwicklungstechniken durchsetzen, wird eingebettete Software immer noch überwiegend händisch implementiert. Damit wird die Softwareentwicklung noch mehr zum Engpass, als sie es ohnehin schon ist, und macht die bei der Hardwareentwicklung erzielten Einsparungen zunichte. Diese Arbeit befasst sich mit dieser Herausforderung und schlägt eine Methodik für den Entwurf eingebetteter Software basierend auf modellgesteuerter Architektur vor, um Kosten zu senken, die Qualität zu erhöhen und die Entwicklung eingebetteter Software zu beschleunigen. Gemeinsam mit dem bestehenden Hardwaremodellierungsansatz wird ein umfassender Generierungsfluss geschaffen, der die abgestimmte Generierung von Hardware und eingebetteter Software gewährleistet.

Der Generierungsfluss schlägt ein Metamodell für eingebettete Software vor, um die Struktur und das Verhalten von eingebetteter Software zu abstrahieren. Durch den holistischen Ansatz wird eine Spezifikation (konzeptionelles Modell) in Instanzen des Metamodells für eingebettete Software und des Hardware-Metamodells (Hardware-Generierungsablauf) übersetzt. Das Ergebnis ist ein gebrauchsfertiger Code, dessen Struktur und Verhalten auf die jeweilige Hardware zugeschnitten ist. Der vorgestellte Ansatz erzeugt den gesamten eingebetteten Software-Stack. Für die HW/SW-Schnittstelle wird ein Automatismus eingesetzt, der das Speicherlayout der Peripheriegeräte optimiert, um eine effiziente Hardware-Abstraktionsschicht zu erzeugen. Für die Treiberschicht durchläuft der Ansatz mehrere Transformationsschritte, um eine Vielzahl verschiedener Treiber-Varianten zu erstellen. Der Anwender kann die Variante gemäß den Systemanforderungen weiter anpassen. Der Generierungsablauf unterstützt (1) hochgradig konfigurierbare Gerätespezifikationen (2) mehrere eingebettete Softwarearchitekturen (3) nicht-funktionale Erweiterungen (z.B. Sicherheitsmaßnahmen) (4) verschiedene Plattformen (z.B. Sprachen).

Das Potenzial dieses Ansatzes für die Industrie ist enorm. Ein Entwickler muss nicht länger ein Experte sein, um z.B., sichere eingebettete Software zu implementieren. Das erspart viel Entwicklungkosten und ist gleichzeitig weniger fehleranfällig, da repetitive Implementierungsaktivitäten entfallen. Außerdem stehen dem Entwickler mehrere Entwurfsalternativen zur Verfügung, um die beste Variante für ein bestimmtes System zu ermitteln.

Acknowledgements

The work conducted in this thesis documents research work carried out at Infineon Technologies AG, Germany, in collaboration with the EDA Chair at the Technical University of Munich, Germany.

First of all, I like to thank my supervisor, Professor Wolfgang Ecker, for giving me the opportunity to write this thesis and for supporting me with ideas, criticism, motivation and guidance. I especially appreciate the numerous discussions about the challenges and capabilities of metamodeling.

Furthermore, I like to thank my mentor Daniel Müller-Gritschneder for very insightful discussions, whose input has been of great importance to this work. I also benefited greatly from working closely with his research group, especially Rafael Stahl and Uzair Sharif.

The SAFE4I and COMPACT projects, funded by the BMBF, have been a great experience that helped me to understand the key challenges of model-based embedded software development and inspired me to make certain research decisions. The collaboration with all members of the funded projects also gave me the opportunity to apply the concepts developed in this thesis to demonstrators.

I am thankful to friends and colleagues who contributed in many different ways: First of all, I would like to express my gratitude to all my bachelor's and master's students who contributed to the development of the tool infrastructure: Nicolas Ojeda Leon, Igli Zeraliu, Andreas Neumeier, Chander Kumar, Mhemed Hajjej and Amaan Jeelani. Second, all colleagues who have participated in Infineon's metamodeling working group: Zhao Han, Lorenzo Servadei, Keerthikumara Devarajegowda, Sebastian Prebeck, Christian Lück and Moomen Chaari. Special thanks to Christian Bernhardt for all the help and support, interesting discussions, and effort to promote and integrate the work's embedded software flow in a very hardware-heavy project.

Working on a doctoral thesis requires not only technical but an equal amount of personal support. Therefore, I would like to express my deepest gratitude to my girlfriend Kristin and our families, who have supported me. Finally, I would like to thank all my friends, especially Andreas Denk, Aditya Deshmukh and the "Dulis" for their support and sympathetic ear in difficult phases of the Ph.D. work.

Contents

Acknowledgements **VII**

Abbreviations **XIII**

1 Introduction **1**
- 1.1 Handling Growing Complexity . . . 1
- 1.2 Outline of the Thesis . . . 4

2 Problem Statement and Contributions **7**
- 2.1 Problem Statement . . . 7
  - 2.1.1 Tool and implementation challenges . . . 8
  - 2.1.2 Domain-related challenges . . . 10
  - 2.1.3 Socio-Technical Challenges . . . 12
- 2.2 Requirements . . . 13
- 2.3 Targeted Approach . . . 18
- 2.4 Summary of the Key Advantages . . . 22

3 Model Driven Engineering **25**
- 3.1 Metamodeling . . . 26
  - 3.1.1 Modeling Language and Domain-specific Language . . . 27
  - 3.1.2 Model Transformations and Code Generators . . . 28
  - 3.1.3 Model Driven Architecture . . . 29
- 3.2 Previous Work . . . 30
  - 3.2.1 Metamodeling environment . . . 30
  - 3.2.2 MDA approach for RTL generation . . . 32
- 3.3 Embedded Software Architecture . . . 33
  - 3.3.1 IP Core . . . 34
  - 3.3.2 Hardware and Software Interface . . . 35
  - 3.3.3 Register Interface . . . 36
  - 3.3.4 Hardware Abstraction Layer . . . 37
  - 3.3.5 Device Driver Layer . . . 38
  - 3.3.6 Application . . . 39
- 3.4 Safety in embedded software . . . 40

4 View Generator and Language Model **43**
- 4.1 View Generator . . . 43

4.1.1 Construction of a View Generator . . . 44
4.1.2 Abstract Language Model . . . 45
4.1.3 Unparser . . . 46
4.2 View Language Description . . . 47
4.2.1 EBNF Distinction . . . 47
4.2.2 VLD Notation . . . 48
4.3 C Code Syntax in VLD . . . 49
4.3.1 MISRA C Compliance . . . 49
4.4 C VLD Constructs . . . 50
4.4.1 File structure . . . 50
4.4.2 Data Types . . . 52
4.4.3 Expressions . . . 54
4.4.4 Statements . . . 56
4.4.5 Documentation . . . 58

5 Register Interface and Hardware Abstraction Layer 59
5.1 Abstract Model of the Register Interface . . . 59
5.1.1 Flexible Memory Layout . . . 61
5.1.2 Bitfield configurations . . . 61
5.1.3 Hardware Access Sequences . . . 62
5.2 Register Interface and HAL Generation . . . 62
5.2.1 Register Interface Hardware . . . 62
5.2.2 Hardware Abstraction Layer . . . 65
5.2.2.1 Implementation Variants . . . 66
5.2.2.1.1 Generic Bitfield Structures . . . 66
5.2.2.1.2 Specific Inline Accesses . . . 67
5.2.3 Access Optimization . . . 67

6 Embedded Software Modeling 71
6.1 Embedded Software Generation Flow . . . 72
6.2 Abstract Embedded Software Model . . . 74
6.2.1 Limitation . . . 75
6.2.2 Objects . . . 76
6.2.3 Activities and Actions . . . 79
6.2.3.1 Object Actions . . . 80
6.2.3.2 Control Flow Actions . . . 82
6.3 Generator Specification Model . . . 83
6.3.1 Organization of the Software Architecture . . . 85
6.3.2 Generator Configuration Settings . . . 86
6.3.2.1 Device Driver Architecture and Design . . . 86
6.3.2.2 Device Driver API . . . 88
6.3.2.3 Safety in Embedded Software . . . 88
6.3.2.4 Analysis and Verification . . . 89

7 Device Specific Generator Frontend **91**
7.1 Template of Embedded Software . . . 92
7.1.1 Device Driver Structure . . . 94
7.1.2 Device Driver Behaviour . . . 96
7.2 Domain Specific Language and Design Pattern Reuse . . . 98
7.2.1 Domain Specific Language . . . 98
7.2.2 Design Pattern . . . 100

8 Device Generic Generator Backend **103**
8.1 Design Decisions . . . 104
8.1.1 Driver Reusability . . . 104
8.1.2 I/O Driver Design . . . 106
8.1.2.1 Synchronous Driver . . . 107
8.1.2.2 Asynchronous Driver - Interrupt Service Routine . . . 108
8.2 Safety Pattern . . . 114
8.2.1 Redundant Register check after modification . . . 115
8.2.2 Watchdog . . . 116
8.2.3 Signature-based Program Flow Monitor . . . 118
8.2.3.1 Model Extensions . . . 119
8.2.3.2 Hardware Requirements . . . 120
8.2.3.3 Intra-procedural PFM . . . 121
8.2.3.4 Instruction-stream PFM . . . 125
8.2.4 Error Handler . . . 127
8.2.5 Results . . . 128
8.2.5.1 Fault Injection . . . 129
8.2.5.2 Diagnostic coverage and overhead . . . 130
8.2.5.2.1 Transformation capabilities . . . 130
8.2.5.2.2 Memory and performance overhead . . . 131
8.2.5.2.3 Diagnostic coverage . . . 131
8.3 Optimization . . . 132
8.3.1 Compiler Optimization . . . 133
8.3.1.1 Compiler integration and training set generation . . . 134
8.3.1.2 Clustering-based compiler exploration in nonlinear optimization space. . . . 135
8.3.1.3 Evaluation . . . 135
8.3.2 Memory layout optimization . . . 137
8.3.2.1 Training set generation . . . 137
8.3.2.2 Register Interface estimation and optimization . . . 138

9 Application **141**
9.1 State Machine . . . 142
9.1.1 Nested if-else FSM . . . 142
9.1.2 Function pointer based FSM . . . 143

9.2 External Software . . . 144
9.3 Analysis of application . . . 145
9.3.1 Firmware Verification . . . 145
9.3.2 Performance Analysis . . . 147
9.3.3 End-to-End analysis flow . . . 147
9.4 Proof of Concept . . . 149
9.4.1 Self Test System-On-Chip . . . 149
9.4.2 Safety Demonstrator - Human-Robot Interaction . . . 150
9.4.3 ML-based Demonstrator - Location Detection . . . 151

10 Summary and Conclusion **153**

Bibliography **174**

Abbreviations

API	Application Programming Interface
BF	Bitfield
CC	Clock Cycle
CFE	Control Flow Error
CFG	Control Flow Graph
CIM	Computational Independent Model
CSR	Control and Status Registers
DSL	Domain-Specific Language
EBNF	Extended Backus–Naur Form
ESW	Embedded Software
EU	Exception Unit
FIFO	First In – First Out
FSM	Finite State Machine
FTL	Fault Tolerance Latency
HAL	Hardware Abstraction Layer
I2C	Inter-Integrated Circuit
I2S	Inter-IC Sound
IC	Interrupt Controller
IP	Intellectual Property
IRQ	Interrupt Request
ISR	Interrupt Service Routine
LUT	Lookup Table
MARTE	Modeling and Analysis of Real-Time and Embedded systems
MBE	Model-Based Engineering
MDA	Model-Driven Architecture
MOF	Meta Object Facility
OMG	Object Management Group
PFM	Program Flow Monitor
PIM	Platform Independent Model
PM	Platform Model
PSM	Platform Specific Model
RI	Register Interface
RMW	Read-Modify-Write
RTL	Register Transfer Level
SLOC	Source Lines of Code

SoC	System-on-a-Chip
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver Transmitter
UML	Unified Modeling Language
VLD	View Language Description
XMI	XML Metadata Interchange

Chapter 1

Introduction

Every generation has to face particular challenges that will influence the lives of tomorrow's generations. The major global challenges of our generation are the climate crisis and digital transformation. Nevertheless, every challenge is also an opportunity for growth and technological advancement. The semiconductor industry is a key driver in tackling these challenges and pushing forward digitalization and decarbonization. From these megatrends, the industry is poised for a decade of growth across multiple end-user verticals, including healthcare, consumer electronics, industrial, automotive, financial services, and retail. According to McKinsey's report [49], the semiconductor industry will witness fast development of emerging technologies to become a $1 trillion industry by the end of this decade.

1.1 Handling Growing Complexity

The rapid growth of the market and the increasing ubiquity of embedded systems in our daily lives are causing a growing productivity gap in the design of embedded systems. The problem is further compounded by the fact that the embedded system's complexity is growing faster than the productivity of system designers. Today's embedded designers need alternative design methodologies to bridge this design gap and cope with stringent costs (see Figure 1.1), time-to-market, and reliability requirements.

Intellectual Property (IP) reuse [74, 167] through modular designs is one paradigm to simplify the design and increase quality and productivity. Nevertheless, in practice, hard-coded IPs suffer from significant drawbacks, as shown in [77]. The major limitation of IP reuse is that a fixed building block is not necessarily best suited for every application and product. Adapting the IP to changing system requirements or the target application requires additional effort and high generality of the IP. This lack of flexibility often negatively impacts the design's performance and quality, which calls for other development strategies. A fundamental approach that is often used to manage complexity is abstraction. In short, abstraction simplifies the design process by hiding all but the relevant details.

Grady Booch correctly points out, "The entire history of software engineering is that of the rise in levels of abstraction" [40]. The first microprocessors were programmed in machine language (*first-generation language*), and even today, all programs must be translated into this language. The code is a sequence of zeros and ones that is difficult for humans to read and even more challenging to develop. The assembly language (*second-generation language*), introduced in 1948, translates human-readable words into a sequence of zeros and ones. Nevertheless, also the as-

sembly language became unsuitable with the growing complexity of the systems. In addition, assembly language is always tied to a specific platform. High-level languages (*third-generation languages*) like Fortran introduce hardware-independency, adding abstract elements such as loops or conditions to simplify the coding of more extensive programs. It enables a problem-oriented representation that is transparent to the developer and meets the demands for user-friendliness and comprehensibility. Subsequently, many other general purpose languages followed, such as C, Pascal or `C++`, which offered a higher degree of abstraction. A general-purpose language is not tailored to a specific domain and therefore requires a detailed description of how to solve a particular problem. Today, the *fourth-generation languages* are intensively researched, which is an upward trend toward higher abstraction.

Model-driven engineering (MDE) and code generation is an approach that raises the level of abstraction to cope with today's design complexity. It enables the development of a fourth-generation language, also called a domain-specific language, tailored to a specific domain. The idea of MDE is to incorporate the aspects of abstraction and restriction in domain-specific languages derived from conceptual models, as noted by [168]. In MDE, a model can represent a specific domain, while code generators can translate the model into a particular target language. This process is called metamodeling and is an excellent way to bridge the design gap. It is applicable in all phases of embedded systems development, such as design, analysis, verification or documentation. The MDE flow systematically translates the conceptual models into a final implementation through different levels of abstraction, adding implementation details to each level. This translation step is accomplished by a generator scheme that handles all possible alternatives of the conceptual model. Rather than reusing hard-coded IPs, model-based development uses generators to create different IP alternatives to optimize the design for a particular application. The consequence of the development process is a shift in focus from code-centric to function-centric and platform-independent engineering. In other words, instead of writing code manually, a designer implements generator templates using scripting languages. He becomes a software developer who must put solutions over technologies and abstraction over code.

MDE and code generation strategies are not new. Ad-hoc code generation from formalisms such as UML (Unified Modeling Language) has been studied for quite a while [57, 141, 197]. Several quantitative and qualitative studies [41, 57] confirm the efficiency and effectiveness of MDE practices in embedded systems development. However, according to Madni and Sievers [132], model-based approaches have not yet fully unfolded their potential in the industry. Studies in [9, 27] revealed the widespread acceptance of informal modeling to facilitate stakeholder discussions. However, the formality of the modeling language is still neglected and not employed for other purposes, such as automatic generation.

MDE and code generation can be applied to various disciplines involved in the SoC development chain, including hardware design, embedded software design, and system and application development. In fact, companies hardly see any benefit in MDE since they employ it only selectively for specific design tasks [45], e.g., architecture design. The full potential, however, only comes to fruition when various engineering disciplines at the process and artifact levels are synchronized coherently. Accordingly, only the consequent use of MDE across the entire design process of an embedded system pays off. Indeed, a holistic approach is a major engineering challenge for the industry to manage the rising development costs shown in Figure 1.1.

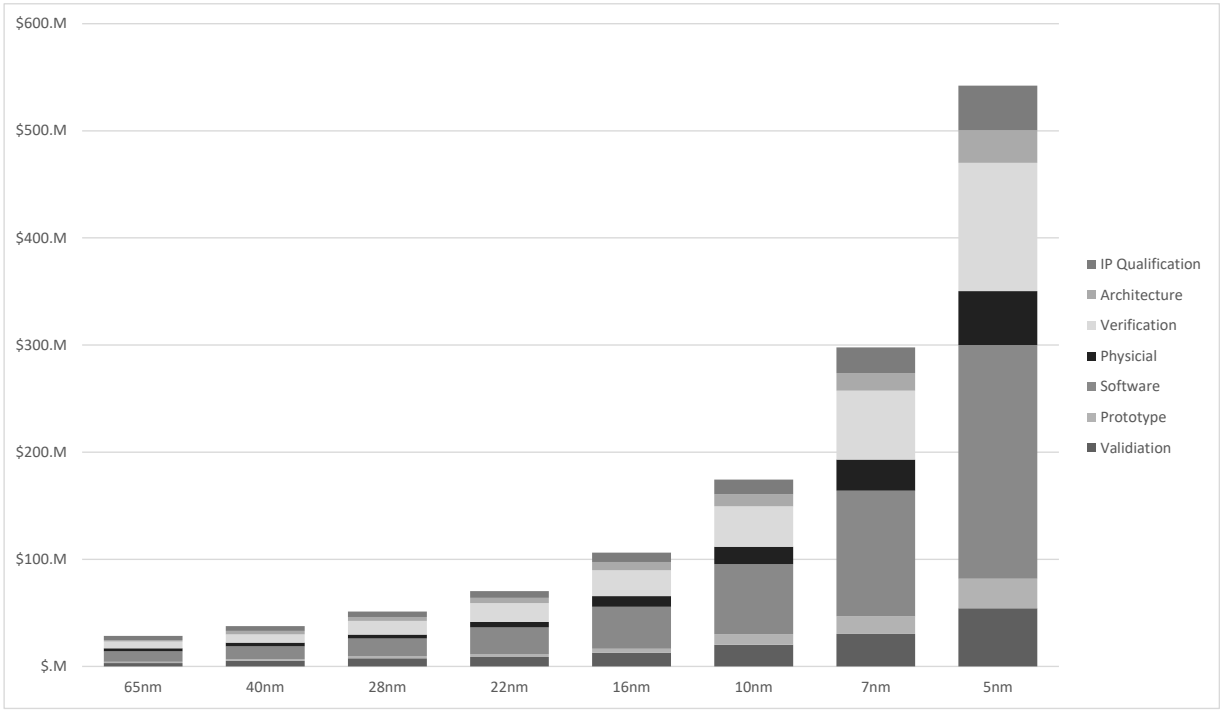


Figure 1.1: *Leading factors driving the SoC design cost for shrinking process nodes. Data source from IBS [91]*

As the graph highlights, the industry can no longer afford to wait for hardware in order to start developing software. Embedded software development is the biggest driver of overall SoC design costs, skyrocketing with increasing process technology. NXP in [1] estimates that tomorrow's car in 2025 requires six times more code than a car in 2018, which counted 150 million lines of code. At the same time, the industry is facing a shortage of software engineers [42], leaving positions unfilled. So the number of lines of code is growing faster than the number of skilled embedded software engineers. The demand for software development is not unjustified. The software stands above the hardware and ultimately contributes to the product's success. Bailey puts it drastically in [25]: "Good hardware without good software is a waste of silicon."

There are three primary reasons why software development cannot keep pace with hardware development:

1. Many semiconductor companies use programmable hardware platforms while relying on software as the prevailing differentiator.

2. Software development teams face challenges that require more engineering expertise to deal with safety, quality or performance.

3. Despite the considerable software effort, design methodologies are still hardware-centric.

Simply adding a software development flow without proper integration is insufficient because hardware and software are closely linked in embedded systems. Moreover, modern development approaches such as MDE are more typical for hardware design but are rarely used in embedded software development. However, what is the benefit of a hardware-only modeling and code gen-

eration approach that partially solves the problems and pushes the bottleneck even more to the software designers?

Many available toolchains are hardware-centric development flows that can not meet system-specific demands. Simply adding a software-development flow without proper integration is insufficient because hardware and software are closely interlinked in embedded systems. Today's industry requires modeling methodologies that minimize effort and maximize efficiency while tackling complex embedded system design challenges. Modern embedded system development demands a holistic approach that enables joint hardware and software modeling. The main principle of such a single-source approach is to capture all relevant information about the system in abstract models. Accordingly, an abstract model takes into account all software and hardware relevant properties of an IP component to ensure interoperability between different generation flows. Based on the separation of concerns, various generation flows can produce different kinds of outputs from that single source.

Any generation flow, e.g., for embedded software, can translate the abstract model via different abstraction layers into the target code. Model-driven architecture (MDA) is one of the most popular approaches, which starts translating the abstract model into a platform-independent model. A platform-independent model specifies a high-level notation of the particular design aspect. Overall, it captures the essential features without considering implementation details, e.g., programming language details. The second translation step in MDA turns the platform-independent model into a platform-specific model before feeding it to the code generator to output the target code.

Such an MDA approach not only automates the development process but also offers a high degree of flexibility for creating different design alternatives in software and hardware. Compared to manual coding, the methodology ensures greater efficiency and stability by automating repetitive, error-prone tasks. Moreover, a holistic generation flow for various aspects opens up many possibilities that pure software or hardware generators do not have. First, it enhances consistency between different design aspects due to the single-source principle, which reduces design errors and improves their convergence. Second, design decisions and their impacts can be evaluated early in development, enabling exhaustive design analysis and optimization. Indeed, such a multi-aspect approach has the potential to bridge the widening design gap in the embedded industry headed by embedded software development.

1.2 Outline of the Thesis

The main driver of this thesis is the increasing design gap that embedded software developer face today. Chapter 2 outlines these challenges of embedded software development and automatic generation approaches. Motivated by these problems, the requirements and the key paradigms for the automatic generation of embedded software are developed. After outlining the targeted approach, Chapter 3 discusses the main principles of MDE and solutions proposed by related work to tackle these challenges. In this context, both the approaches used in this thesis and divergent approaches are discussed. The main contributions of this work are presented in the following chapters.

First, Chapter 4 introduces the concept of a generator to create code generators. The approach is demonstrated for a Misra-C-compliant code generator. Chapter 5 deals with the hardware-software interface, the lowest layer of the embedded software stack. The interface enables interaction between hardware and embedded software. For the SoC's efficiency, an innovative metamodel and generator of the hardware-software interface are presented to enable tuning of the register layout.

The layer above the register interface is the device driver layer. The following chapters describe the highly configurable embedded software generation flow capable of auto-generating embedded software layers such as the driver layer. It consists of two decoupled generator steps and a domain-specific model that exploits embedded software's typical pattern. Chapter 6 starts by giving a detailed introduction to the complete framework. It introduces a new domain-specific model for embedded software design. In addition, it outlines concepts to increase the flexibility and design variability of the framework. Next, Chapter 7 describes the first transformation layer, the generator front end, which assembles an abstract embedded software model from a specification of the IP. This chapter introduces the "generator language", the domain-specific language used to assemble the structure and behavior of the embedded software model. It demonstrates the language's use through various IP driver examples and is intended as a guide for deploying other IP driver generators.

Chapter 8 introduces the heart of the framework, the second transformation layer, also called the generator backend. The generator backend is hidden from the user configurations and specifies numerous generic transformations that can be applied across various IPs. Three types of transformations are presented in this chapter:

- Transformations to produce different design and architecture alternatives such as the I/O driver design.
- Safety transformations automatically incorporate safety patterns, e.g., program flow monitors, into the design.
- Automatic optimization step that identifies better design alternatives, e.g., memory layout, based on a pre-defined cost function.

Chapter 9 introduces the last layer of the embedded software stack, the application layer. It demonstrates how the framework can also be exploited to create applications. Moreover, industrial applications are presented that have been built using this framework. Finally, Chapter 10 concludes the thesis and discusses future research topics.

Chapter 2

Problem Statement and Contributions

One of the major challenges in today's software development is the shortening of development cycles and, at the same time, the increase of software complexity. On top of that, embedded software is subject to increasingly stringent requirements in terms of power, performance and safety. In order to meet these challenges, the semiconductor industry is experiencing a paradigm shift in software development from manually coding to model-driven engineering (MDE).

According to the survey in [9], the industry perceives shortened development cycles and cost savings as the main reasons for using MDE. Additionally, reliability, compatibility, quality improvement and maintainability are generally considered to be essential advantages of MDE. In theory, a model-based approach is a perfect concept for developing complex heterogeneous systems, as it provides the ability for abstraction in the development process, from requirements development through design, implementation, integration and verification. However, the introduction of MDE in the industry entails major challenges that affect established development processes and organizational structures. Along with the need for new tools and methods, engineers must also change their mindset and acquire the skills required to apply MDE

Model-driven engineering has been used with considerable success in some embedded system development disciplines. However, it has not yet been widely applied in the embedded software community. This chapter outlines the key challenges in model-based embedded software development that must be overcome to get acceptance and to achieve high cost and time savings. These challenges are grouped into three categories as defined by Bucchiarone et al. in [48]: Tooling and Implementation Challenges, Domain-Based Challenges, and Socio-Technical Challenges. The requirements, solutions and expected contributions for a new methodology are developed based on the identified problem space. Finally, the work's approach is introduced, drawing on the previous findings.

2.1 Problem Statement

Embedded systems and especially embedded software are becoming increasingly complex, and hence traditional design concepts reach their limits in efficiency. Abstraction is a proven approach to managing complexity. It increases the comprehensibility of complex systems at the expense of the additional effort required to maintain abstract views. This section classifies the challenges of a model-based approach based on abstraction principles.

The challenges are grouped into three categories, defined by [48], that reflect the problem areas of State-of-the-Art model-driven development. One group of these are the tooling and implementation challenges that address model-based development's conceptual and theoretical aspects. Indeed, the lack of high-qualitative and industry-appropriate tools is often quoted as one of the main aspects hindering the adoption of MDE. Also, introducing new model-based instruments is associated with more complications in practice than considered in theory. Thus, expectations are not met.

The domain-related challenges characterize the second category. Embedded software development is fraught with specific issues that every embedded development team must deal with. For this reason, embedded software development must be entrusted to professionals with extensive experience. An embedded software modeling framework created by experts for everyone should overcome these difficulties. Generally, a stable, qualitative and reliable generation is necessary for each domain. For the embedded software domain, further three basic requirements for the generated software must be fulfilled: Small memory footprint in terms of the stack size, ROM and RAM usage, low power consumption and high performance. As a rule, focusing on one of all these requirements compromises the others. Ideally, the generation framework aims for a wise compromise between all these cost parameters. With the increasing complexity of embedded systems, safety and security concerns are also rapidly increasing. An embedded software tool must provide safety and security measures to comply with the standards that must be adhered to.

In fact, apart from the technical issues, the social and community challenges are one of the key factors for the success of MDE. These challenges are classified as socio-technical challenges. MDE has not yet gained widespread acceptance in the embedded software community. The socio-technical challenges primarily concern the apparent task of encouraging traditional developers and development teams to use and accept MDE.

2.1.1 Tool and implementation challenges

According to [9, 126], the lack of good tooling is widely reported as one of the main aspects hampering MDE adoption. Many tools have usability issues, such as difficulties with tool customization, tool integration, or lack of code generation capabilities. Others are simple but only cover a limited scope that is insufficient for modeling complex industrial embedded systems. Especially in the embedded systems domain, the size and diversity of artifacts, including models, metamodels and their transformations, is enormous. Accordingly, configurability must be preserved while complexity is not shifted to the tool infrastructure. In other words, the complexity of the MDE approach needs to be tamed or hidden from the user.

When implementing an MDE tool, it is important to consider that various stakeholders are involved in the development process of an embedded system. Some stakeholders use models as blueprints only for communication and documentation purposes. Others consider models to be the key artifacts of the development process. A fundamental difference between the two use cases is that there are increased demands on the completeness and accuracy of the models during development. According to Bucchiarone et al. in [28], it requires an efficient and reliable mechanism for mapping abstract models to more concrete models or the target code. In order to enhance reliability, the modeling flow has to assure consistency across different transformation layers. First, all

models and the modeling languages must be consistent with the associated metamodels. Second, all transformations have to be deterministic, which helps to manage the consistency of models undergoing modifications.

One potential factor that drives the adoption of model-based development is an intuitive textual modeling language[1] while handling the code as a model. So it lowers the entry barrier for a modeling tool. Such a textual modeling language is derived from a metamodel. It is used to create, read or modify an instance of the metamodel. In essence, it is the central language that the developer uses to implement the generators. In order to come up with a suitable modeling language, generator and platform, the developers need to have a good understanding of the domain for which they develop the infrastructure [215]. In fact, many tools provide a modeling language that misses flexibility since it is customized for a certain metamodel. In practice, the development of a mature infrastructure happens gradually over a relatively long period of time, based on experiences and various projects. Building a fully comprehensive framework from scratch is practically impossible. Indeed, modifications and extensions on the metamodel with new features must be reflected on the modeling language. So, tools with missing customizability require much effort to redesign the modeling language.

The usability of model-based development tools for managing models of complex systems is often described as a challenge. In general, model-driven approaches try to cope with the system complexity by increasing the level of abstraction. The main problems of many modeling tools are bound to the model abstractions at different stages of the development process. Finding a suitable model abstraction in a self-explaining notation used to describe different heterogeneous views is the path to the tool's success. However, it is always necessary to keep in mind that each level of abstraction adds a further transformation layer, which requires extra maintenance effort. The number of layers and the degree of abstraction between these layers must be chosen wisely. A good modeling tool must find a suitable intermediate solution to tame the complexity of the systems but also preserve variability while minimizing the effort of implementing transformation templates. As a rule, the models' size is less of a problem than the variety of artifacts that increase the effort required to develop transformations and generators. A transformation converts a model at one level of abstraction into a model or code at a lower level. A generator may consist of multiple transformation steps that transform a model across multiple levels of abstraction.

Instead of manual coding, a designer in the model-driven approach has to implement the transformation scripts of the generator. The effort required to implement the generators is a crucial criterion in the acceptance of model-based solutions since it demonstrates the advantages over manual coding. A rough method to compare the manual effort required to implement the generator or target code is to measure the software metric, e.g., source lines of code. Actually, many tools require far more effort[2] compared to traditional coding, quickly leading to designers' frustration. Finding the right balance between simplicity and expressiveness of the modeling language used to program the generators is crucial to the tool's success.

[1] A modeling language is any graphical or textual language that provisions the construction of models following a consistent set of rules.

[2] Indeed, this is an unfair comparison since generators can produce many alternatives. A proper comparison should also consider the number of variants that can be generated.

A good modeling language and generator design mainly depend on the correct analysis of the models involved in the transformation, as stated in [46, 48]. The challenge lies in analyzing the correspondences between the artifacts of different models and understanding the semantics of such connections to create a good modeling language that ensures traceability from requirements to implementation. For further simplification, the modeling language should use a standardized notation that adopts the features of modern object-oriented programming languages. In addition, many modeling languages lack common software development practices such as information hiding and DRY (Don't repeat yourself) to ease the use of the language.

The study in [140] demonstrates that using model-driven development in the industry can lead to quality improvements but may also cause productivity losses if designed improperly. Besides modeling complexity, usability issues and user-friendliness, insufficient toolchains and the use of MDE with legacy systems are often reported as challenges. As a result, the maturity of most tool environments is considered inadequate for large-scale deployment in industry. This is often caused by the lack of customizability of the toolchain. Certainly, it is nearly impossible to develop an MDE tool from scratch that takes into account all possible industrial applications. Consequently, developing a tool that grows and stabilizes with the number of applications realized is even more important. According to many surveys [48, 126, 207], customizability and extensibility are the most requested features by many practitioners to be improved by tool vendors. There are no limitations in an intuitive toolchain since users can extend or modify all aspects of the tool, including model-to-model generators, code generators, metamodels, and the modeling language.

Obviously, the key factor, apart from the tool's limitations and complexity, is the generated implementations' quality and variability. Essentially, the challenge and goal are to produce code of similar quality to manually written code. A major challenge, but also the primary attractiveness of MDE, is the enormous variability of target implementations that can be automatically generated. These implementation alternatives result from different design decisions that the generator resolves when constructing different implementations. A good generator makes cost-related design decisions to match the target code to the requirements and the application. A designer relies on each generated implementation, so a crucial requirement for MDE tools is to guarantee the code's correctness and quality. On the one hand, the tool must include features to verify the generator and the implementation alternatives. On the other hand, the tool must ensure consistency between implementation and requirements.

2.1.2 Domain-related challenges

Developing complex and portable embedded software requires a clear software architecture. The software architecture is the blueprint that manages large-scale embedded systems' design, development and maintenance. It embodies the fundamental organization of a system in its components, their relationships to each other and to the environment [102].In general, software architecture specifies layers with contractually defined interfaces as functional boundaries between components. Such an interface provides declarations and function prototypes that separate the application from the low-level code. Accordingly, a layered software architecture improves portability, allowing both layers to be reused in other applications or on other hardware [32].

A modeling framework has to reflect the structure of the software architecture and must be able to generate these various layers. There are few tools that offer a solution for the generation of all layers. Often, embedded software generators focus only on individual software architecture layers, such as the hardware abstraction layer. A tool that covers both the application and the low-level software development has many advantages and is in high demand in the industry. Primarily, such tools help to overcome the problems encountered by many users concerning synchronization between the different engineering disciplines and software layer development processes.

As mentioned earlier, a major challenge for tools is their ability to create modeling languages for different domains. In order to come up with a suitable embedded software generation infrastructure, the developer needs to have a good understanding of all embedded software challenges. As discussed before, one of the biggest challenges in this respect is finding an adequate degree of abstraction and detail for the embedded software model. In the end, the model should contain all features to support different target platforms and languages, such as C or Rust. Similarly, a proper generation framework should be able to generate the entire embedded software avoiding the need for manual post-processing. Many tools only create the skeleton of the embedded software, more or less the header files. This basic generator also reduces work, but the core behavior still has to be implemented manually. Such a partial-generation approach has several drawbacks since each variant must be modified before it can be further processed, e.g., tested or analyzed. A generator that requires manual rework squanders much of the potential of model-based development. However, a good modeling flow for embedded software should create embedded software that can be directly processed and compiled. The abstract model of the embedded software must therefore contain all artifacts required to create the design. This implies artifacts to represent the skeleton and the embedded software's behavior.

Embedded software developers face numerous challenges when manually creating software or using model-based development tools. In the following, the fundamental challenges for embedded software are listed. A big challenge arises with the growing system complexity. A modeling tool must deal with simple projects and scale with complex applications from various domains (e.g., IoT, automotive). Furthermore, it must be able to handle various designs, e.g., processor families, which range from simple and extremely power-saving processors to highly powerful and configurable processors. An embedded software generation framework needs to adhere to these design constraints and be adaptable to specific hardware definitions.

Many tools for generating embedded software simply ignore hardware requirements. Instead of generating hardware-specific designs, they aim for a high degree of modularity in the generated designs. Although this approach can support any hardware platform, it also has significant drawbacks. The implementations are not optimized for resources, and thus, the quality of a manual hardware-specific implementation cannot be achieved. Embedded software can be implemented in a resource-efficient and high-performance manner if hardware properties are mapped correctly to software decisions.

The design can also be evaluated and optimized using this hardware-software coding approach. According to [226], optimization is basically seen as a cycle of design, evaluation and redesign. This is a major challenge as hardware and software elements interact in complex ways, making choosing the best combination difficult. So it goes beyond the scope of many generation systems. On the one hand, it requires an interface to simulate the generated designs and analyze their

results. On the other hand, an approach is necessary for randomizing the configuration of design alternatives.

A large proportion of the development costs are attributable to the burning issue of increasing safety and security requirements in products. Safety and Security are features that enhance an application with life-saving or protective functionality. Generally, it is associated with strict requirements that demand more design expertise. As stated by Streitferdt et al. in [194], modeling tools barely address safety-related aspects in models or their transformations. For this reason, the challenge is integrating safety-certified model transformations into the generation flow. Ideally, these should be derived from best-practice knowledge of safety measures used in existing projects and products.

2.1.3 Socio-Technical Challenges

When seeking solutions to technical challenges and developing an MDE ecosystem, there is an urgent need to consider social aspects. Indeed, a new toolchain is only accepted and adopted in practice if all stakeholders validate and approve it. For this reason, the social and community challenges become critical factors for the success of MDE tools.

Model-based development is intended to be the catalyst that enables domain experts to develop the tools they need in their domains with their knowledge. In manual coding, this knowledge, such as best practices, might get lost, but with explicitly defined transformations, this knowledge is preserved in the transformations - accessible to a community. In this way, even developers with less experience gain access to an infrastructure that produces high-quality implementations. Krogstie defines this principle in [117] fittingly as "Modeling by the people, for the people". Ideally, anyone, even a non-technical person, should easily use the model-based development framework to create any design. Moreover, they can even explore a vast design space. This truly might be a great vision for the future and an important impulse to overcome the challenges of embedded development processes.

However, most developers do not dare to move from traditional to model-based development. Of course, this skepticism is strongly influenced by the technical and domain-specific problems encountered so far, but a general reluctance to accept model-based development is also noticeable. One reason is the missing know-how about the concepts and tools. Especially for traditional embedded software developers, the principles of metamodeling can be a culture shock that does not lead to acceptance.

For designers, the transition generally means adopting a completely different way of thinking and developing. Instead of focusing on the implementation of one variant, the designer of generators must consider all possible variants while the framework builds this variation. This way of thinking is one reason why metamodeling is considered more challenging and quickly triggers the designer's frustration. A good tool, however, can contribute a lot to relieve the designer, e.g., by taking over design decisions. The report in [96] supports these findings and highlights three essential factors for the success of model-based development. These include an iterative and progressive approach, organizational commitment, and motivated users. Many surveys also suggest the concept of design-by-example, which is intended to slowly introduce the designer to the new metamodeling approach and its details [177]. Ultimately, though, as noted in [2], "the problem

with a completely new programming paradigm is not learning a new language... The tricky part is learning to think in a different way."

Particularly in industry, the acceptance is low. Establishing such a new development process is associated with costs and enormous efforts. First, it requires a high effort to train developers. Second, the new tooling infrastructure must be integrated into legacy processes, an issue with many tools. Thirdly and most significantly, the benefits of a transition to metamodeling only become apparent in the long run. In the beginning, the development of generators is costly. With increasing generator reuse, the development costs amortize and the company profits enormously.

2.2 Requirements

Overall, the step from manual coding to generator-based development depends on the use case. The benefit of generators depends on the required level of flexibility. Keep in mind that developing a model-based infrastructure is a long-term commitment that requires continuous improvement to compete with emerging applications and challenges. Before implementing a modeling framework, it is important to consider the potential savings. Are the flexibility loss and the effort required to implement a new approach justified?

This thesis definitely answers with "yes" to the question for the embedded software domain. A good framework avoids typical design errors and can speed up development, thus reducing costs in software development. The three identified categories of challenges capture the typical issues, user concerns, and desires for model-based development frameworks for embedded software. Based on these challenges, the requirements for a new approach can be derived. By adhering to these, typical pitfalls can be avoided, and the full potential of model-based development can be exploited. Thus, a real productivity boost can be achieved compared to existing tools.

The elaborated requirements concern the overall framework but also the requirements of the sub-components, such as the code generator or the different abstraction layers. In the following, these requirements are outlined. In the next section, their implementation and key advantages in the target approach are discussed.

Today's SoC complexity puts high demands on tools and engineers. In particular, the complexity of embedded software has increased massively in recent years. A modern modeling approach for the generation of embedded software has to solve the technical, domain-specific and socio-technical challenges to establish itself in the industry. Building on the given problems, the requirements for a new embedded software modeling and generation flow can be formulated.

The framework presented in section 3.2 from previous work allows the development of model-driven generator toolchains with little effort. Applying this framework to the hardware domain helps to model the complete hardware of embedded systems. However, an embedded system also includes embedded software that is closely synchronized with the hardware. In order to establish this modeling framework for building embedded software in an efficient and user-friendly way, the following requirements must be met.

R 1 *Single-Source Principle:* Embedded software modeling must be performed as part of a holistic system modeling framework. Modeling and generating embedded software isolated from hardware modeling is fragile and prone to errors. A holistic system modeling approach en-

sures consistency, quality, and correct and efficient interoperability between hardware and software. Accordingly, the generation flow for hardware and embedded software should proceed from joint abstract models.

R 2 *Separation of Concerns:* A new SoC modeling tool should have mechanisms that support both vertical and horizontal separation of concerns. Modularization of the functionality through separation of concerns helps to develop large applications quickly and efficiently. It reduces the complexity of dealing with large specifications.

R 3 *Reusability:* A new modeling framework must simplify the coding effort and eliminate the need for designers with platform-specific expertise. In other words, the new modeling methodology has to hide implementation details on the abstract layers. Furthermore, the generation approach for embedded software should follow the same pattern as the hardware generation flow. Similarly, the generator languages for hardware and embedded software modeling should follow the same characteristics. The new modeling framework should consider both aspects. It should be designed in a generic way to decrease the learning effort required to apply it across different aspects of the overall embedded system development chain.

R 4 *Consistent Modeling:* The tool must ensure that artifacts are generated consistently. So, it has to support consistent modeling by following the constraints of all involved metamodels. An instance is considered (syntactically) consistent with a metamodel when it satisfies the conditions specified in the metamodel. In general, the model consistency depends strongly on the degree of integration of the metamodels into the toolchain.

R 5 *Correctness by Construction:* A new tool for modeling and generating embedded systems must ensure the correctness of the design. Correct-by-Construction minimizes development effort on debugging and verification. By ensuring correctness-by-construction, the designer can generate, execute, and analyze models directly.

R 6 *Extensibility:* It must be possible to extend the framework with new functions. A new modeling framework should be extensible for capturing new IP devices, code generators, and domain-specific features, e.g., design patterns.

R 7 *Graphical Notation:* All applied metamodels rely on a graphical notation to express the design visually.

Code generation is an essential aspect of model-based development. It translates the abstract model into target languages. One way to accomplish this is directly translating the abstract model into the targeted language. A better approach, however, considers the requirements R4, R5, and R6 using an intermediate language model. This language model is first assembled before being translated (model-to-text transformation) into the target code. The following requirements must apply to the code generation and associated language models.

R 8 *Target Languages:* A large number of different languages are in use in SoC development. Meanwhile, existing languages are being replaced by modern languages. Accordingly, a

modeling framework needs to be extensible with new language generators from time to time to support new output languages, e.g., Rust and MicroPython.

Respectively, an intuitive approach for generating language generators is needed. Such an automatism should build the language models and the model-to-text transformation scripts for any kind of formal language. Thus, various target languages can be supported in a new modeling flow to cover different platforms.

R 9 *Meta Syntax:* The language generator approach must support any formal language known from RTL and embedded software design. The language generators must be automatically derived from a metasyntax of a formal language. The metasyntax represents a metamodel that constrains the syntax of the underlying formal language, thus reducing syntactic errors.

R 10 *Coding Standard:* Apart from the correct syntax, the generated code must also comply with the coding guidelines that promote the safety and reliability of software for embedded systems such as MISRA C.

R 11 *Readability:* In addition to the formal language syntax, the generated code must also have good readability. A language generator must, therefore, also consider formatting, e.g., indentations. This way, a designer can customize the coding style by adapting the code generators.

R 12 *Documentation:* The generated code must be adequately documented to ease subsequent post-processing, integration, and analysis. Consequently, the language generator needs to include appropriate elements for documentation.

An essential factor for the new methodology's success is the quality of the generated embedded software code. In particular, whether the methodology can solve the challenges and problems encountered in developing embedded software. Design metrics provide important insight in determining the quality of the embedded software for the respective hardware.

A design metric is a measurable characteristic of the system, e.g., performance, memory footprint, safety or power consumption. Typically, these are conflicting requirements, i.e., optimizing one requirement may negatively impact another. For example, a more powerful embedded software may demand more memory. In the following, the most crucial design metrics are introduced.

R 13 *Memory Footprint:* Embedded software has to cope with limited memory in many application areas (e.g., IoT, security controllers). For this reason, a generation framework must be able to build code that is at least as compact as manually written code. In this context, various aspects must be considered and optimized: static and dynamic memory, instruction and data memory.

R 14 *Performance:* Of course, the new embedded software modeling framework must ensure that the generated code executes at least as fast as the manually written code.

R 15 *Functional Safety:* A new modeling tool must provide features to create safe and reliable software that complies with ISO standards such as ISO-26262.

R 16 *Security:* A new modeling tool must be capable of building secure embedded software. It should contain features to protect data and the interfaces from being abused.

In order to address the requirements R13, R14, R15 and R16 the following rules have to be fulfilled for the embedded software modeling flow:

R 17 *Abstraction and Concretization:* The new abstract model for embedded software should be close to the specification and the target code. If the metamodel is too close to the specification, the generators need to cover every model instance of this metamodel. This significantly raises the development effort for generators and their transformation scripts since the generators have to examine the model extensively. In contrast, when the metamodel is close to the target code, the development effort is reduced at the expense of reusability and the degree of automation.

R 18 *Flexibility:* The modeling framework for generating embedded software must be highly flexible. First, it needs to support an extensive range of IP and system configuration alternatives, respectively hardware variants. Second, the framework has to provide a wide range of design choices to tune the embedded software for specific hardware implementation.

R 19 *Hardware Adaption:* The generated software must be optimized for the IP components' specification and functionality, respectively, to its hardware implementation. The generated code should not include any features that might not be required for specific functionality.

The abstract model that characterizes the embedded software design is the most important component for modeling embedded software. The domain-specific language used for implementing the transformation scripts is derived from this model. A transformation script transforms the specification of an embedded system component into an instance of the abstract embedded software model. The following requirements should apply to the transformations and the abstract model.

R 20 *Legacy Code:* The framework must be capable of maintaining and transforming legacy code into an abstract model.

R 21 *Behavior and Structure:* The platform-independent embedded software model must include all features necessary to auto-generate the fully functional device driver. The platform-independent embedded software model must include all features necessary to auto-generate the fully functional device driver. Consequently, the model has to contain both features to represent structures, e.g., interface definitions, and features to model the behavior of the data and control flow.

R 22 *Portability:* The platform-independent embedded software model should be sufficiently generic that it can be mapped to different target platforms and languages. According to R8, various code generators can thus be interfaced with the abstract model.

R 23 *Software Stack:* The platform-independent model for embedded software must be capable of specifying different layers of the embedded software stack.

The success of the new methodology depends on its entry barrier. One of the biggest struggles in introducing a generation framework is the development of the transformation scripts. Writing generators capable of creating thousands of variants instead of manually implementing the code for a single variant is challenging and requires a shift in the developer's mindset. Therefore, it is even more important that the framework is straightforward and ensures that certain design decisions remain hidden from the user. The following requirements are crucial for improving this entry barrier:

R 24 *Simplicity:* In order to obtain acceptance from designers and entire project teams, a new modeling tool has to be straightforward to use. Implementing a generator should not significantly exceed the effort required to implement the code manually. As an evaluation criterion, the number of lines of code can be considered analogous to the effort.

R 25 *Design Pattern:* The framework should provide a design pattern library. As specified by the modeling domain, a design pattern describes a transformation rule that assembles or modifies a subset of the abstract model. These are intended as extensions to the automatically generated domain-specific language. A designer can draw on these design patterns or further extend the library as needed.

R 26 *Information Hiding:* The principle of information hiding is a central guideline for modularizing complex software systems. Information hiding, also called encapsulation, seeks to hide design decisions or implementation details from the user. The framework should hide design decisions from the user that automatism can solve. In addition, irrelevant implementation details should be hidden to make changes to the underlying platform easier.

R 27 *Hardware Interface:* The generator designer must not worry about the hardware and software interface details. The framework should automatically derive the interface, including the register mapping.

No matter how easy to use, a framework will not gain acceptance if the quality of the generated code is poor. The created software must also be subjected to quality assurance measures for quality and functional safety.

R 28 *Design Validation:* The correctness of the generated code must be automatically verified in the generator backend of the modeling framework. The modeling tool must establish a handshake with the compiler and the simulator to ensure the functional integrity of the code. A compiler can check the code's syntactic, logical or semantic correctness. The simulator, in turn, validates the functional correctness of the code running on the generated RTL.

R 29 *Safety Assurance:* Safety-critical systems must be able to handle faults without causing damage or misbehavior. An automatic approach is necessary, which evaluates the reliability of the auto-generated safety measures and fault handling methods. Fault injection is a technique widely used that introduces faults into the system during simulation. The goal of fault injection is to introduce faults into the software or hardware to ensure that the system still fulfills requirements while these faults are present.

One of the main arguments in favor of a modeling framework over traditional design is, obviously, the ability to build various design alternatives R18. This diversity of variants enables design exploration of the variant's design metrics (trade-off analysis) and, thus, design optimization. This way, appropriate design decisions for each application can be identified.

R 30 *Design Exploration:* A new modeling tool that generates different design alternatives should also analyze design metrics such as those design alternatives' performance or memory requirements. This way, a large data set is ready for further analysis, e.g., through machine learning.

R 31 *Optimization:* A modeling framework must determine the best configuration from an extensive range of alternatives based on a pre-defined cost function.

2.3 Targeted Approach

This thesis presents a solution for a top-down generation flow for embedded software that satisfies the previously discussed requirements. The methodology extends the hardware generation framework that will be described in Section 3.2 with a software complement to provide a comprehensive model-driven SoC development toolchain. This embedded software generation framework can fulfill all the needs in developing complex distributed embedded software. To this end, it covers all phases of modeling, analysis and validation up to the automatic generation of the implementation.

The basis of MDE is to capture the requirements of a system in a specification model. This model is then gradually extended by implementation details by systematic model elaboration. In this way, the RTL implementation of an SoC is derived from system's specification. The primary driver for this approach is the fact that a design is not determined by the hardware alone. An efficient embedded system is based on the software interacting with the hardware in the best possible way. The decisive factor in this approach is to consider hardware and software holistically since both are mutually dependent. As hardware-software co-design suggests, the proposed generation framework provides a top-down generation flow that follows R1 (Single-Source Principle). So, the tool considers the same specification models as inputs for the holistic generation of hardware and embedded software, as shown in Figure 3.5. The single-source design principle ensures optimized, consistent and correct generation of hardware and embedded software according to the requirements of R5 (Correctness by Construction).

In order to use a single model for multiple purposes, the tool executes a multi-domain generation flow starting from the same models. Therefore, the thesis extends the existing hardware generation flow with a parallel embedded software generation flow. The total embedded system generation is generally structured into two MDA flows, each implementing a different domain generator. Each MDA flow specifies a domain-specific mapping from the high-level system properties to the hardware or embedded software. Such a formalized mapping enables the investigation of design decisions and their impact on the software and hardware. So, it facilitates the selection of the best combination of software design and underlying hardware. Both domain generators are implemented with the same python-based modeling infrastructure R3 (Reusability) and code generator approach. This reduces the maintenance and training effort. Modeling skills and techniques are independent of the object being modeled. A good modeler can use the same skills and

techniques to model the hardware and the embedded software (assuming they have the appropriate domain knowledge). A user can transfer knowledge between domains, lowering the entry barrier and simplifying communication between domains.

As a second key principle, besides the single-source principle, the proposed SoC generation framework applies the principle of R2 (Separation of Concerns) supporting both vertical and horizontal separation of concerns. MDA proposes a vertical separation of concerns mechanism consisting of three layers of abstraction and two model-to-model generator stages. As proposed by MDA, the first layer specifies the characteristics of an IP of the embedded system in individual models, such as a processor, memory controller, or communication IP model. The second layer specifies the abstract design of the domain, e.g., the abstract embedded software. The last layer adds implementation details to the abstract model. In order to resolve the vertical separation, a series of two generator stages, namely an IP-specific generator frontend and an IP-generic generator backend, are introduced. The generator frontend maps the IP specification to an embedded software model, which specifies the general behavior of the IP. The backend converts this model into a language model comprising all details.

The horizontal separation follows the organization of the embedded system into different subcomponents. This way, different concerns (IPs) are explicitly separated. Each of them is characterized by its model and dedicated embedded software and HW generator, which makes them explicit and thus more traceable, easier to change, and potentially reusable. Adding a new IP component to the tool is straightforward because the investment in automation has already been made. Once a new IP component needs to be supported, the developer only needs to develop the IP-specific frontend generator to satisfy the IP's capabilities. Meanwhile, the IP-generic part of the tool can be reused R3. Both concepts, single-source principle and separation of concerns, are essential features to reduce the generator development effort and increase the design quality.

The proposed framework considers R4 as it provides an intuitive and automatically generated infrastructure that ensures consistency and validity of models across the generation flow. Among others, an automatically generated modeling language is provided, which is used to assemble instances of the metamodel, the so-called domain-specific languages (DSL). In other words, an instance is developed in a language defined by the instance's metamodel at a higher meta-level. Accordingly, an instance of the metamodel is always valid as it complies with the modeling language. Since the entire infrastructure for a particular metamodel can be generated, the proposed approach fulfills R6 (Extensibility). Extending the features of the metamodel directly affects the modeling language and infrastructure. So features can be added anytime as long as they do not modify the previous structure of the metamodel. In addition, building a new modeling infrastructure for a new IP component or code generator is straightforward, thanks to this intuitive framework. The only requirement for building the infrastructure is that each metamodel must follow the Unified Modeling Language (UML). This satisfies R7, offering a standardized, graphically based formalism for capturing system models.

Another advantage of the proposed model-driven framework for embedded software is its code generator approach. It proposes an innovative concept in Chapter 4 to automatically build code generators. So, various target languages and coding styles can be created as required by R8 (Target languages). For this purpose, the thesis introduces a new language named VLD (View Language Description), which maps the meta-syntax as demanded by R9 of a formal language

similar to EBNF. Compared to other meta-syntax languages, VLD adds formatting elements (tabs, spaces, line feeds, empty lines, comments). This way, the generated code is syntactically correct and formatted for better R11 (Readability). As an example, this thesis introduces the C-VLD used to build the C-Generator, which contains elements to fulfill R10 (Coding Standard) and R12 (Documentation). The C generator restrains the C language to a subset to be MISRA C compliant and to simplify the code generator. Additionally, the C-VLD provides terminal elements that describe specially formatted comments, which can be extracted by a documentation generator such as Doxygen. Chapter 6 introduces the main components of the proposed framework, the platform-independent embedded software metamodel, and the two generator steps. Moreover, the chapter highlights the main strength of the approach, the capability to build a wide range of variants, as demanded in R18 (Flexiblity). This is accomplished by following two approaches. First, the framework features highly configurable specification models of IPs that enable the construction of a wide range of IP designs, including HW and embedded software. Second, the framework specifies embedded-software-specific parameters to customize and tune the two generator steps.

The central model of the approach is the embedded software model, which fulfills R22 (Portability) abstracting various implementation languages and platforms. It contains all the artifacts necessary to design the R21 (Behavior and Structure) of the complete embedded R23 (Software Stack). The structure of the abstract embedded software model matches R17 (Abstraction and Concretization), providing a good balance between implementation details and abstraction to increase flexibility and reduce the coding effort for generators. In summary, the embedded software metamodel is significantly more abstract than the implementation code that would have to be developed manually otherwise. This way, different manifestations, e.g., architecture and design alternatives, can be created from a single embedded software instance. The architecture of a software system is the result of all design decisions, which can hardly be modified when coded manually. One striking example of this approach is the register access handling R27, described generically on the abstract level. However, other elements of the embedded software model are challenging in terms of abstraction and remain close to their implementation, e.g., type and object definitions. The proposed framework follows the principle of abstracting as much as possible unless the complexity increases drastically.

The first step of the generation process is handled by the IP-specific generator frontend that assembles the embedded software model according to the requirements. Following the single-source principle, the generator aims to improve the design to be compatible with the hardware and to satisfy the design metrics R13 (Memory Footprint) and R14 (Performance). In other words, only those elements are designed that are necessary for the particular application and the underlying hardware R19 (Hardware Adaption). Thus no dead code is produced. In order to support as many hardware alternatives as possible, embedded software is often constructed in a generic nature using design patterns. However, the disadvantage of using design patterns is that the designs may not be able to solve specific problems optimally on the cost of design metrics. In the framework context, a design pattern describes a parameterizable transformation rule (or software factory) applied as a wrapper to an artifact of the abstract embedded software model. Different IP generators can reuse a design pattern to facilitate the design of recurring processes (sub-behaviors). For this purpose, it assembles a substructure of the embedded software model concerning specific design requirements. Developing generators using design patterns can provide many benefits, such as

improved productivity, quality and evolution capability. So far, the framework proposes a large library of R25 (Design Patterns) to simplify generator coding complexity.

The IP-generic generator backend constructs the final target code from the abstract embedded software model. The main idea of the backend is to resolve certain design decisions hidden from the developer R26. The hardware-software interface R27 is a great example of hiding design decisions. Depending on the interface's memory layout, the hardware area's performance and register access may improve or deteriorate. For example, a compact layout has less performance (inefficient read and write accesses) but also requires less design area. The proposed framework presents an automated approach that examines different memory layouts to extract the ones offering the best cost trade-off. Further examples are provided in Chapter 8.

The backend further realizes a style-based architecture refinement through iterative horizontal transformations[3] where the source and target model reside in the same abstraction layer. This way, an abstract architecture can be mapped into numerous design alternatives, such as an interrupt-based or polling design. Following the same strategy, the backend also improves the non-functional metrics, such as the design's R15 (Functional Safety) and R16 (Security) of the design. These transformations inject safety measures as safety patterns to extend the embedded software with non-functional artifacts. So far, the framework provides a list of industrial-proven safety measures that ensure the temporal and logical execution order, e.g., watchdog or program flow monitors. Following the given safety pattern principle, a designer can extend the library with any further safety and security patterns.

Numerous design alternatives can be generated using this model-driven approach. The quality and reliability of each generated design is an important issue in the acceptance of the tool. Therefore, the framework extends the generation process with a subsequent optional validation and exploration step that examines different design alternatives. The extension includes a compilation and simulation step described in Chapter 9. The framework uses *GCC* as a compiler and *Verilator* or *Xcelium* to run the simulation. The entire embedded system generation flow for the simulation is executed to build the complete RISC-V-based SoC on which the binary is executed. Note that the proposed approach does not execute a detailed verification but provides immediate user feedback on the functional correctness and executability R28.

Furthermore, a simulator-based fault injection concept that extends *Verilator* with fault injection capability is introduced. So, the functional safety, the system's reliability, and the safety pattern efficiency are evaluated as required by R29 (Safety Assurance).

The simulation result and the compiled binary are used to determine the design metrics for each alternative, such as memory footprint, performance, or hardware area. As a result, the designer obtains a vast data set mapping costs to the design specification. Indeed, such a large data set opens the door for R30 (Design Exploration), e.g., through machine learning or other trade-off analysis techniques. The thesis provides two examples for design exploration and R31 (Optimization). One approach to compiler flag optimization and one to hardware-software interface memory layout optimization.

[3] A horizontal transformation can also be called an endogenous transformation. It is a transformation between models expressed by the same metamodel.

2.4 Summary of the Key Advantages

The presented approach provides several advantages for developing embedded software or embedded systems. The following list briefly explains these:

- Developer productivity is improved because repetitive aspects do not need to be coded manually repeatedly. A designer simply modifies the high-level requirements to create different designs.
- The code generator approach is faster than humans and can handle optimization complexity that far exceeds human cognitive abilities without being as error-prone. So the framework helps to deal with optimization challenges.
- Models serve as a better basis for discussion and help to improve the understanding of the systems at the design level.
- Models are very close to the problem domain and omit implementation details that are irrelevant for understanding the logical behavior. Thus, they reduce the semantic gap between the concepts and the implemented solution.
- The framework allows experts to capture their knowledge in design patterns and transformations. So they share their expertise with other members of the organization. If well documented, this knowledge remains in the patterns and transformation even if the experts leave the organization.
- The modeling framework follows a definition of a stringent software and system architecture. This reduces errors and improves the consistency and quality of the software architecture.
- With code generation, the code quality is improved. A template ensures correctness-by-construction and high-quality code compared to manually written code.
- The generated code is very readable and includes comments and other artifacts to improve readability. Compared to manually written code, it is easier to understand as it follows production rules that define the formal language's syntax and formatting.
- The approach separates different concerns and system components explicitly. Each one is modeled, making them explicit and thus potentially reusable, more traceable and easier to modify.
- The mapping of the models to the implementation code is deterministic. This means that the design metrics of the implementation are known to some extent. It is a great advantage for optimizing the embedded system based on cost analysis.
- The framework is extendable in all respects. Any code generators and sub-systems can be included as long as they adhere to the separation of concerns. Likewise, existing models can be extended unless their original structure is changed.

- Embedded software generation is tightly connected to hardware generation. Both start from a common specification level. As a result, the embedded software is well adapted and efficiently leverages the hardware resources. Manual coding, in contrast, is often generic and inefficient.
- The effort for generator coding is similar to manual coding. However, manual coding is significantly more expensive if the effort is scaled according to the potential design alternatives that can be generated. Accordingly, the return on investment from using this approach increases each time it is reused.

Chapter 3

Model Driven Engineering

A model reduces the system's complexity providing a more appropriate view of a system. In general, models can express systems' properties, structure, and behavior in various scientific disciplines, such as all areas of mathematics and science. A well-known model is the climate model [70], which is used to predict the climate. It is built on a set of mathematical equations that describe the physical laws that govern the behavior of the atmosphere and oceans. Similarly, a model can also express the behavior and structure of embedded systems.

In software engineering, a model serves as a specification that gives an overall picture of the system under study. It defines and organizes the information that is needed to develop a system. Accordingly, models make project planning more efficient and facilitate communication between stakeholders. However, models are not just used for the documentation but can also serve as primary artifacts for the software development process. Using models as recurring patterns within a software development methodology can increase productivity and compatibility between systems. This principle is considered model-driven engineering (MDE)[4]. According to France and Rumpe [75], "MDE is typically used to describe software development approaches in which abstract models of software systems are created and systematically transformed to concrete implementations." Respectively, MDE-based approaches include various techniques such as metamodeling, model transformation, and code generation. MDE narrows the gap between an abstract description and the target code. It thus contributes to the streamlining of development processes.

MMDA is a specific MDE proposal premised on the idea that a single model is too fuzzy and cluttered for designing a larger and more complex system. Consequently, MDA employs multiple models that describe the system from different viewpoints or at different levels of abstraction. In particular, it specifies viewpoints representing the system's requirements, design and implementation. This separation of concerns makes MDA a popular modeling approach, as it achieves both platform independence and language and system independence. For example, an MDA approach allows developers to focus on domain concerns rather than dealing with platform-specific details such as application programming interfaces or language guidelines [75].

In the following, the basic concepts of MDE and MDA are outlined. Based on these concepts, this chapter presents the prior work elaborating on the MDA principle to introduce an industrial model-driven RTL generation flow. Furthermore, this chapter discusses the state of the art of MDE in embedded systems and embedded software development.

[4]In the literature, also synonyms such as model-based engineering (MBE) model-driven development (MDD) or model-driven software development (MDSD) are also widely accepted. In this thesis, the notion of model-driven engineering is preserved.

3.1 Metamodeling

Modeling is the process of building models that transform a perceived view of the system into an abstract representation. According to Stachowiak [189], a model is a simplified image of reality characterized by three fundamental properties. (1) *Mapping criteria:* models do not capture all system features but only those relevant to the designer. (2) *Reduction criteria:* models do not capture all features of the system, but only those that are relevant to the designer. (3) *Pragmatism criteria:* models are tailored to the needs of the intended use.

Selic [176] emphasizes the criterion of reduction and abstraction as almost the only means of coping with emerging complexity. Hiding details irrelevant for a viewpoint helps to clarify the system's essence. Good abstraction is difficult to achieve and should be chosen wisely, not forced [116, 176]. Intelligent abstraction reduces complexity, increases understandability, and allows the prediction of interesting but non-obvious properties of the modeled system. Above all, a model must be economical, i.e., it must be much cheaper to construct and analyze systems [176].

Metamodeling (literally "beyond modeling") is the act of modeling models [188]. It is a crucial concept in MDE approaches as they provide a mechanism for the automated development of well-structured and maintainable systems. A metamodel defines the permissible structure that must be satisfied by a set of models [191, 193]. Compared to a model, a metamodel is an abstract representation of the model itself [82]. In MDE, a metamodel specifies abstract syntax (the constructs and relationships) of a modeling language to express a model. A model created with the modeling language is called an instance[5].

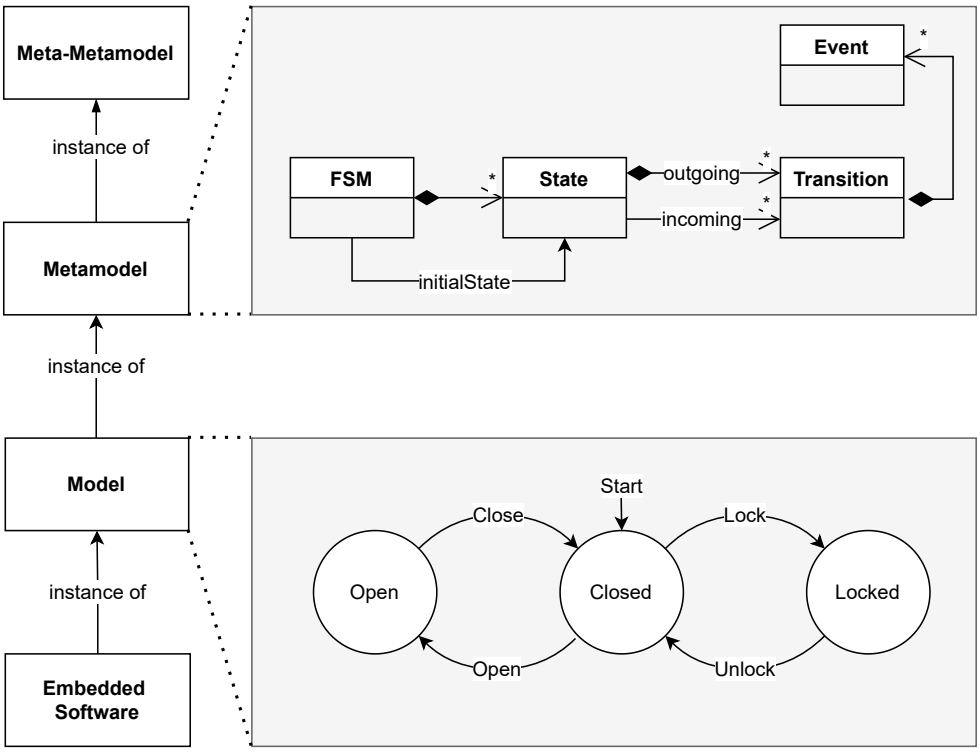


Figure 3.1: *The hierarchy between meta-metamodel, metamodel, model and system under study.*

[5]In the following, the terms instance and model are not distinguished. Both designate an element that complies with the rules of the metamodel.

To continue the train of thought: A metamodel is a model describing the modeling language. Thus there also exists a meta-metamodel. Figure 3.1 depict the hierarchical relation between meta-metamodel, metamodel, model and the system (e.g., hardware circuit or embedded software). The Object Management Group (OMG) also proposes this layered infrastructure, which employs the Meta Object Facility (MOF) as a meta-metamodel. MOF provides the language for creating UML metamodels. A UML metamodel defines a "certain aspect" of the system, such as the finite state machine (FSM). In addition, the metamodel supplies the language to build any type of FSM. An FSM model provides an abstract view that can be reused for various purposes, for example, code generation of RTL circuits or embedded software.

An MDE infrastructure relying on these modeling layers and, in particular, on a common meta-metamodel can be generically processed in a unified manner. Bézivin calls this principle in [34] "the unification power of models". Exploiting this generic nature is the key driver for increasing the efficiency of MDE, as it enables automatic model operations such as code generation, transformations, and analysis techniques.

3.1.1 Modeling Language and Domain-specific Language

Metamodeling is the process of creating a domain-specific model by explicitly defining its constructs and rules. A domain-specific language (DSL[6]) can formally describe the structure, behavior, and requirements of a specific system of this domain. This allows a designer or modeler to work directly with the domain concepts at an abstract level since the DSL follows the abstraction and semantics of the domain. According to Kleppe, a domain-specific modeling language comprises three main aspects [113]:

- the domain concepts and rules (abstract syntax)
- the notation used to express these concepts (concrete syntax)
- the semantics of the language.

The abstract syntax of a DSL is specified by a metamodel, which contains well-formed rules to constrain all instances that can be created. Consequently, the metamodel provides the formalized specification of the language; in other words, the metamodel represents the grammar of the DSL. The concrete syntax of a DSL represents the human-readable notation that represents the abstract syntax in a graphical or textual notation. In general, the concrete syntax maps the metamodel's concepts to the visual or textual representation of the metamodel. The graphical concrete syntax establishes links between concepts and visual symbols, while the textual concrete syntax establishes links to the syntactic structures of the language. Such an analogy is well-known in formal languages characterized by a metasyntax notation such as EBNF. Similar to the textual concrete syntax of a metamodel, a metasyntax notation reflects the permissible structure and composition of expressions and statements of a programming language. Indeed, a good modeling tool features a concrete textual syntax for creating and modifying instances and the corresponding graphical concrete syntax for visualizing them.

[6]This thesis considers the two terminologies, domain-specific modeling language and domain-specific language, equivalent.

While the concrete syntax seeks to leverage correct interpretability, the semantics of the language adds meaning to the modeling language [57]. Bryant et al. observe a lack of a standardized method for defining semantics [47]. For this reason, a slightly different approach is commonly used in MDE to introduce translational semantics. Translational semantics comprises a mapping between abstract syntax elements and a specific platform providing well-defined and executable semantics. As pointed out by [47], such a mapping is represented by hard-coded model interpreters, e.g., code generators that produce executable source code from models.

3.1.2 Model Transformations and Code Generators

The central concept of MDE is to specify the system as a model on an abstract level. The transition from this abstraction level to a concrete implementation is achieved by model transformations, which play a crucial role since they automate complex, tedious, error-prone and repetitive software development tasks [29]. Figure 3.2 illustrates a model transformation. It is an automated process that constructs a target model from a source model according to a transformation definition.

The generator executes this transformation process; a script that specifies the transformation definition is called the generator template or transformation template. Templates specify a set of transformation rules on how constructs of the source language can be transformed into constructs in the target language [56]. The transformation rules are built with the DSL. In this context, the generator can map any instance of the source metamodel to an instance of the target metamodel, assuming that the instances follow the semantics (sem) of the metamodel.

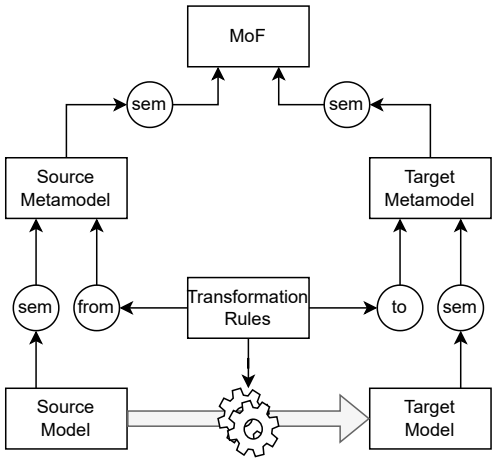


Figure 3.2: *Metamodeling and model-to-model mapping [35].*

Essentially, two types of transformations are distinguished in MDE. On the one hand, model-to-model transformations translate models that are both subject to a metamodel. On the other hand, model-to-text transformations or code generation produce source code from models. Typically, these transformations are mostly vertically oriented (vertical transformation), refining a view from higher to lower levels of abstraction. Instead, a horizontal transformation[7] derives a view at the same level of abstraction.

[7]The horizontal transformation is distinguished between an endogenous and an exogenous transformation. An endogenous transformation is a restructuring between two instances of the same metamodel. An exogenous transformation is a transformation between instances of different metamodels.

3.1.3 Model Driven Architecture

Model Driven Architecture (MDA) is a concept for model-driven, generative software development proposed by the Object Management Group (OMG) [120, 136]. It is a particular form of model-driven engineering focusing primarily on the unambiguously incremental refinement of models into platform-specific code. It aims to manage software technologies' and platforms' complexity by capturing the entire software development process in models [44] and separating domain concerns from implementation concerns. The involved models define all levels of development, from requirements analysis to implementation of the target system, automating the transformation of abstract models into executable systems. MDA encourages the use of three successive models, Computational Independent Model (CIM), Platform Independent Model (PIM) and PSM (Platform Specific Model), to implement hardware and software systems. Figure 3.3 illustrates this MDA principle as Y-chart [120].

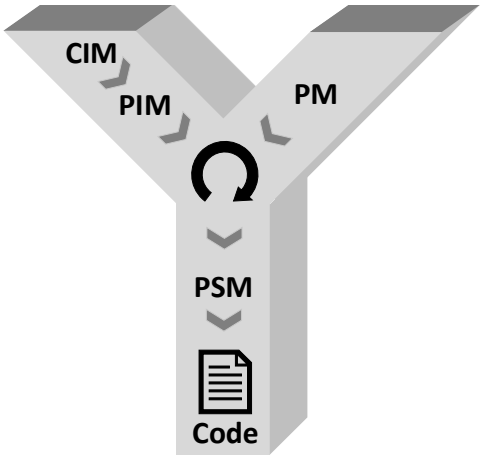


Figure 3.3: *Y-Chart of the MDA principle [120].*

CIM: The CIM captures the requirements of an application in a formal specification independent of the system's realization. It characterizes the software or system's domain and the "conceptualization perspective" [185]. Hence it is also called domain or business model. As the name suggests, the CIM reflects an informational view rather than the actual implementation. In the following, the terminology "specification" is used for CIM.

PIM: The PIM describes the structure and functionality of the system regardless of the implementation technology or platform. It builds on the CIM model and adds information about the processing of the planned software system. To achieve platform independence, the PIM uses language-neutral implementation semantics on a high level of abstraction to describe the system's behavior. This description can be realized in several specific platforms (e.g., programming language, development environment).

PSM: The PSM adds platform-specific aspects and relevant technical details for implementing the PIM. The properties of a PSM are determined by the platform model (PM). In this way, the implementation is adapted to the properties of the respective target platform. Finally, the PSM model contains all details necessary for the automated generation of code (model-to-code transformation).

PM: A platform model is the representation of a platform at the model level. So, it specifies the technical details relevant to implement the PIM.

MDA identifies model transformation and code generation as fundamental mechanisms to facilitate software development and migration. It proposes a sequence of two model-to-model transformations that iteratively add details to derive a PSM from the CIM. Subsequently, the derived PSM is translated into the final implementation using the code generator. Abstraction and transformations in MDA ensure a high degree of reusability, variability and maintainability of the generated program code. Overall, this is intended to render software development efficiency, cost-effectiveness, and quality.

This MDA principle has been adapted for RTL generation as described in Section 3.2. In this thesis, an MDA principle for embedded software generation is presented. As the central model, the FW-PIM is presented. As the RTL flow, the MDA approach for embedded software introduces horizontal model-to-model transformations to incorporate non-functional add-on features such as safety measures.

3.2 Previous Work

3.2.1 Metamodeling environment

The Unified Modeling Language[8] (UML) is the best-known graphical language used to model object-oriented systems. Overall, UML diagrams specify relationships between objects to constraint, design and document system artifacts. Indeed, UML is not a programming language, but some tools utilize UML diagrams to generate domain-specific development environments. One of these tools is *Metagen*, introduced in [66, 67], which has since been used successfully in the industry for various design processes and designs. *Metagen* provides a generic approach that automatically generates a Python-based modeling and code generation infrastructure.

The purpose of *Metagen* is to overcome the challenges of introducing metamodeling in various design domains. It accelerates the transition from manual coding to generator-based design. The approach follows a single-source strategy that generates all essential components of a domain-specific modeling infrastructure from one UML metamodel, including its modeling language or domain-specific language (DSL). A metamodel captures the abstract syntax of a modeling language, including all its structures, and thus provides clear interfaces. Consequently, when assembling an instance of the metamodel, it is necessary to apply the underlying concepts defined in its metamodel. So, all components of the framework must be compliant with the metamodel. In conclusion, *Metagen* specifies an automatism that derives the Python-based modeling infrastructure from the metamodel's metadata.

Figure 3.4 illustrates the structure of a template-based generation framework created by *Metagen*. A metamodel, the root of the concept, specifies all possible IP variants, such as here all alternatives of an I2C peripheral. Based on this I2C metamodel, the generic approach automatically derives an I2C-dependent modeling language (I2C API). The language is a mapping of the metamodel's structure into a Python class library. The I2C API provides a set of functions to

[8] When speaking of UML, implicit derivatives such as SysML and MARTE are included.

access and populate various I2C instances, e.g., *setter* or *getter*. This way, an instance of I2C can be expressed both in an XMI format and by the Python class library. The API handles the translation between both formats by parsers (Reader and Writer).

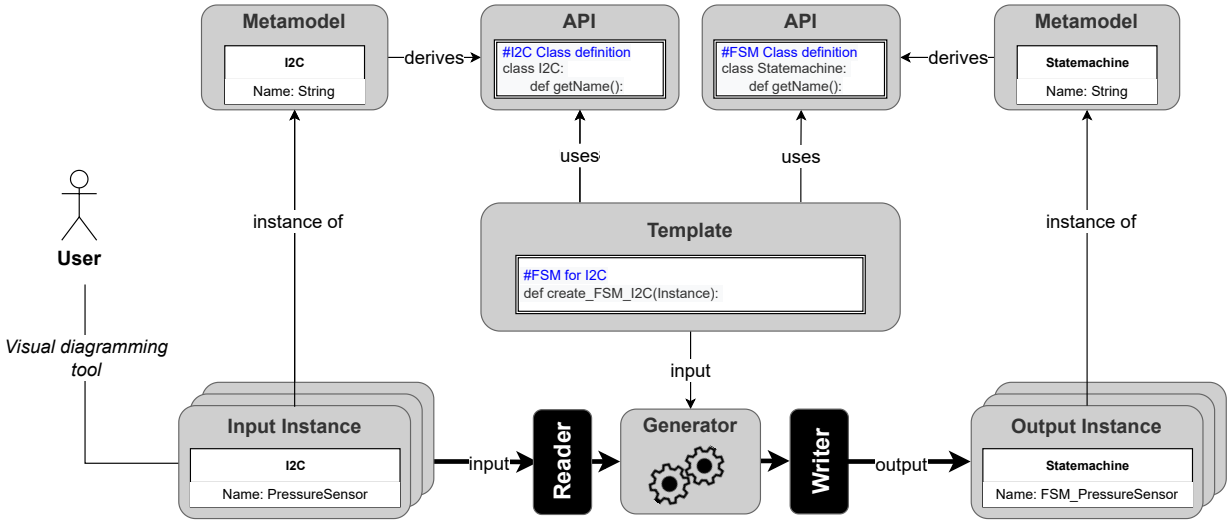


Figure 3.4: *Components of a template-based generation framework.*

Due to the clear interfaces and the polymorphic language, this modeling environment simplifies the implementation of template engines. A template engine, also considered a generator, uses the auto-generated API as a DSL to describe model transformation rules (generator tasks). A model transformation maps a model as input to a new software aspect (e.g., code). Principally, model transformation can be classified into model-to-text or model-to-model transformation. Model-to-text transformations are code generators translating models directly into a target language, e.g., VHDL, C or LaTeX. A model-to-model transformation, as shown in Figure 3.4, is a model refinement involving a target modeling environment. The designer of a model-to-model transformation template uses the API of both environments to transform the input model into an output model. Continuing the previous example, the designer once implements a generator that translates all possible I2C variants into instances of, e.g., a state machine. When designing a system, the designer can reuse the generator and only needs to configure instances of the I2C, e.g., with a visual diagramming tool such as Enterprise Architect, to create the appropriate state machine.

In summary, *Metagen's* generic generation of modeling frameworks perfectly solves the problem of SoC development and all its design aspects. The generic generation approach based on the single source strategy combats the concept of hiding [108] in metamodels and improves consistency from the beginning. Complexity details are moved to the automatically generated API and thus remain hidden from the user when defining transformations. In addition, due to the generic flow, there is a semantic correspondence between the notations of different modeling languages built with *Metagen*. This allows the designer to transfer design concepts he is familiar with from one modeling flow to another, reducing the effort required to adopt a new modeling language.

3.2.2 MDA approach for RTL generation

In [64, 171, 172], this metamodeling environment is applied to establish an MDA-inspired approach to automate digital hardware design. As MDA proposes, detailed in Section 3.1.3, the complexity of code generation is handled by a sequence of model transformations that translate an abstract model into a more concrete model. The lower part of Figure 3.5 shows such a three-layered sequence for a hardware generation framework. The hardware development flow from specification to implementation (HDL code) is formalized and divided into models of different abstraction. So, transformations automatically generate the hardware design by adding implementation details across MDA's three abstraction layers.

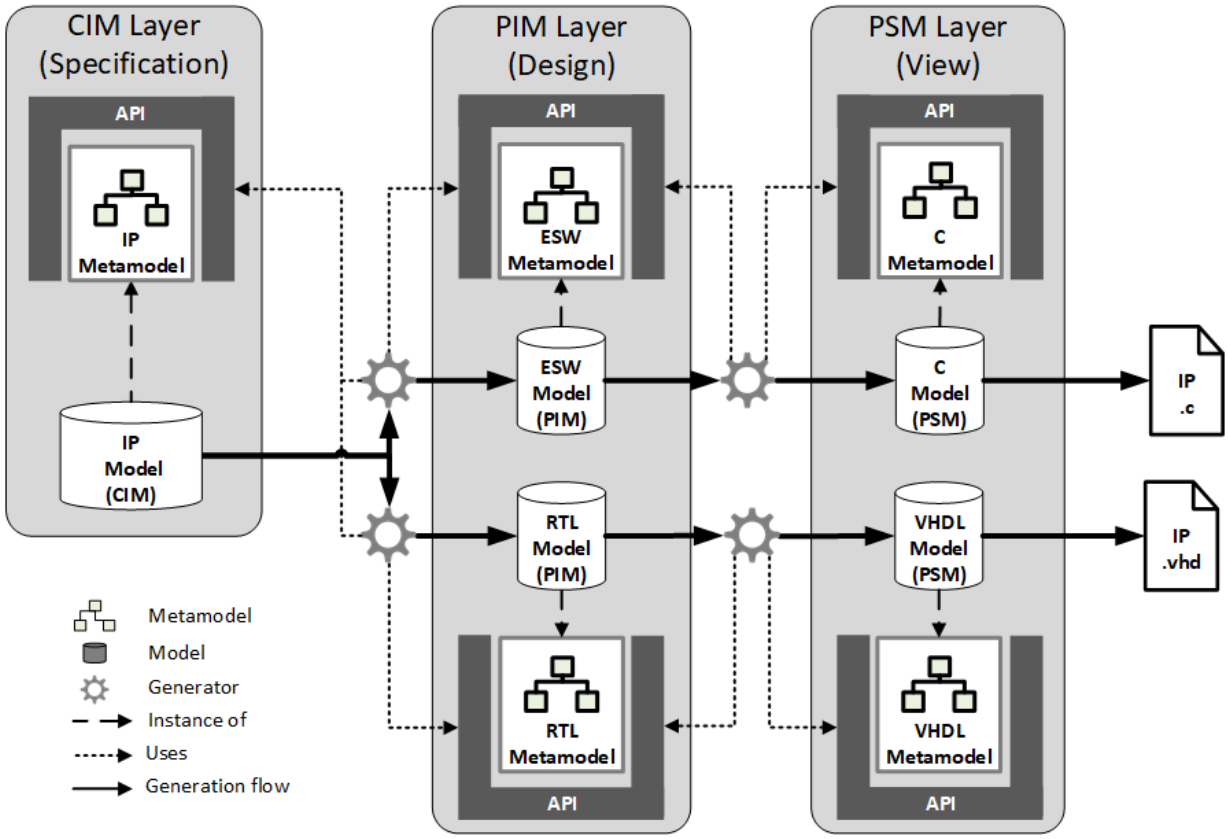


Figure 3.5: *A framework for building IP generators following the three-layered MDA approach.*

The first layer defines the properties and requirements of an IP or subsystem, e.g., I2C, timer or GPIO, in a formal specification of an *IP Model*. Initially, the designer can choose the scope of features from a set of high-level definitions provided by the IP Metamodel. The high-level requirements are transformed into an abstract *RTL model* (domain-specific model) that represents an intermediate design of the RTL microarchitecture. It captures the RTL design through HW primitives (e.g., AND) and building blocks (e.g., FSM) provided by the *RTL Metamodel*. The *RTL Metamodel* and its automatically generated python-based hardware design language are introduced in [85, 171].

A second transformation maps the *RTL Model* to a platform tailored to the language of the target view. For example, one *RTL Model* can be mapped to different implementation models, such as a *VHDL Model* or *SystemC Model*. A *VHDL Model* defines an abstract syntax tree of

VHDL, constrained by the VHDL Metamodel, which specifies the metasyntax of VHDL. Finally, such a language model can be translated into the target output.

In addition, by embedding domain knowledge in a high-level language, designers can leverage modern software techniques from object orientation and dataflow programming to design hardware. Python language, in particular, is ideally suited for this purpose, as it has the high flexibility of a scripting language and an efficient syntax. The designer can use all features, such as functional programming, polymorphism and operator overloading, within the model-to-model transformations. Consequently, the Python-based approach lowers the barrier for hardware designers to adopt a metamodeling framework in hardware design. According to [171], this is a significant advantage compared to other frameworks. For example, the "Eclipse Modeling Framework" [193] is more powerful but also complex and thus very difficult to learn and adopt for hardware engineers.

In a nutshell, this approach simplified end-to-end hardware flow, automates the generation flow, but can also generate many IP hardware implementation alternatives. However, an IP or complete embedded system without the necessary embedded software support is not of great value. This thesis addresses this shortcoming and introduces an automatic code generation flow that runs parallel to the hardware generation.

3.3 Embedded Software Architecture

An embedded system consists of a hardware platform and embedded software. It is constructed using reusable IP blocks. The architecture of an embedded system can be divided into several layers. These layers make it easier to deal with the growing complexity of embedded systems. It defines fixed boundaries that decouple low-level operations, such as register accesses from high-level behavior. In this way, implementation details can be hidden at the application level. For example, the application code does not need to know all details of the driver. There are many different approaches to how to divide these layers. In [33], some concepts are discussed, e.g., 2-layered, 3-layered or 4-layered architecture. In this work, however, the embedded software architecture is divided as shown in Figure 3.6.

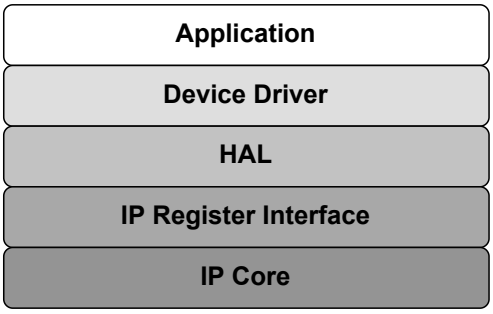


Figure 3.6: *A layered view of the embedded system architecture.*

The lowest layer is the hardware layer containing the IP core (also called the peripheral), which performs the operations, and the hardware/software interface, also called the register interface. The register interface provides a set of memory-mapped registers to control the execution of the IP but also to enable interaction between hardware and software.

The accesses to these registers, which are necessary to control the hardware, are defined in the first software layer, the hardware abstraction layer (HAL[9]). On the one hand, it offers direct macros for writing and reading individual bit fields and registers. On the other hand, it offers more complex access sequences to multiple registers and bit fields combined in a single function call. The HAL keeps details like memory layout or register permissions hidden from the higher embedded software layers, and a designer does not have to worry about bit fields and registers. The device driver sits above the HAL and provides hardware functions with a standardized interface. This way, the driver layer fully abstracts the hardware and acts as an intermediate layer between the hardware device and the application. For example, a serial communication IP driver provides a generic interface that is the same for all hardware configurations that always features a "receive" and a "transmit" function. Finally, the application layer[10] can access these functions without knowing the exact hardware being used.

In the following, modeling approaches from related work are introduced that have automated or partially automated these layers.

3.3.1 IP Core

In electronics design, an IP core is a functional block of a stand-alone RTL unit that can be reused in various systems. An SoC contains several IPs, e.g., a communication controller, memory cores, crypto cores, and arithmetic cores. In most IP design generation approaches, the IP is formally captured in a data structure, while simple print statements or templates with macros handle the HDL code generation. A better approach is to extend established programming languages with hardware-specific primitives to increase the level of abstraction in hardware design.

A language widely used in academia that facilitates the creation and reuse of RTL designs for IPs is *Chisel*. *Chisel* [24] is a hardware construction language embedded in Scala for describing digital logic designs. *FIRRTL* [125] is the intermediate language empowering *Chisel* with a generic hardware design description. This intermediate language can be utilized to build different highly-parameterized hardware generators to produce complex RTL architectures in Verilog. For example, the rocket chip generator in [17] uses *Chisel* to generate cores, caches and interconnects of a SOC.

The survey of Käyrä and Hämäläinen provides a list of other hardware construction languages besides *Chisel* in [110]. For many hardware generators, Python is the preferred language of choice as an HDL language due to its simplicity and clarity in popularity. The most popular Python-based HDL languages are *MyHDL* [59], *PyHDL* [84] and *PyMTL* [130].

All these languages alone cannot solve the problem of generating IPs. A generator approach must use a formal specification of the IP to construct the HDL language. Such ad hoc RTL generation from formalisms such as UML has been studied for some time [118, 159, 214]. A complete generation approach that turns IP requirements into RTL code is enabled by the MDA flow described in 3.2.2.

[9] In some other layered architectures, the HAL is defined differently and is placed above the device driver. The definition in this thesis is that only the HAL can access the hardware. All other software layers can only access hardware through the HAL.

[10] The application layer can be composed of a pure application and a runtime environment/operating system.

***Differentiation from related work**: First, languages like Chisel only partially solve today's design problems since it is a pure hardware design generator and provides no firmware support. So it does not fulfill R1 and R4. Furthermore, Chisel does not start with a formal specification. Compared to our approach, it loses the link to specification and has no underlying formal semantics.*

3.3.2 Hardware and Software Interface

The communication between an IP and the CPU or memory controller is realized through the IP-included hardware and software interface. A hardware and software interface connects an IP of the SoC to the top-level system via the bus interface. In addition, each IP is controlled via a set of memory-mapped registers accessible via the bus. The hardware-software interface specifies the bus interface and register layout. Registers of such an interface are also called Control and Status Registers[11] (CSRs). A CSR appears for the software layer as an accessible storage location that allows the embedded software to interact with the hardware. Each CSR may also provide a side-effect behavior compared to ordinary memory. This side-effect may be triggered when reading or writing the particular register. A register interface is a component frequently occurring in a system and always has the same structure. It only differs in the memory layout and the bus interface. The manual implementation of the interface is a tedious and repetitive task that is well suited for using metamodeling and code generation [100]. A metamodel of a register interface mitigates the problems encountered and can serve as a single source for the automatic generation of all types of views, software, RTL, and documentation [65].

A number of human-readable source formats are available that help specify the register interface and manage the register layout. Nevertheless, two unified standards provided by Accellera have prevailed in the industry: SystemRDL [97] and IP-XACT [4]. The metadata of both standards provides a similar range of features that can be used to configure, verify, and integrate IPs into advanced systems. On the one hand, IP-XACT is an XML format that focuses more on the interconnection of IP interfaces featuring different bus protocols and bus definitions. It supports a simple extension mechanism to add additional vendor-dependent design and flow information. SystemRDL, on the other hand, is a language that contains more syntax features for describing registers but is not as easy to extend as it does not follow the XML schema.

Both standards provide a common vendor-independent representation of a register interface to enable automatic configuration and integration through different automation tools. Many IP suppliers and electronic design automation (EDA) vendors have committed to this standard to increase portability and drive agile design reuse and automation. Indeed, the standards only specify the important features of the register interface but not how the features are translated into a particular view. So, it is not recommending any generator. A large variety of commercial and open-source generators/tools are available that take SystemRDL or IP-XACT to create synthesizable RTL code, UVM register models, documentation or C header files.

Some basic open source projects are *RgGen*. [202] and *PeakRDL*. [142], which provide generators for different views but only support a limited number of SystemRDL/IP-XACT features. More advanced open-source tools are the *RegisterTool* [147] of *OpenTitan*, *Kactus2* [106] from the Tampere university, or the Open Register Design tool [144] released by Juniper Networks. Com-

[11] Other names for CSR, such as Special Function Register (SFR), are also commonly used.

pared to the mentioned open source tools, commercial tools usually offer generators capable of building more target views and languages. They can mostly handle both standard formats while supporting the complete feature set (e.g., various AHB bus protocols). Most importantly, they are more reliable and produce high-quality and industry-standard codes. The most frequently applied tools are the *CSRCompiler*. [5], *DesignSpec* [26] or *Magillem* [15].

***Differentiation from related work**: The register interface (RI) metamodel used in this work specifies a RISC-V-specific bus interface for atomic instructions, embedded software-related extensions, and some additional functions to describe registers. Most importantly, the RI metamodel specifies a decoupled composition of registers and bit fields that allows optimization of the memory layout.*

3.3.3 Register Interface

Implementing the RTL of a RI, including the address decoding and the registers with different characteristics, is a defective and weeks-long effort for a designer. Generating the register interface's RTL code (VHDL, SystemVerilog or SystemC) from SystemRDL or IP-XACT is more or less straightforward. It results in a significant improvement of the RTL development process. One of those tools supporting the generation is the open-source tool *Kactus2* [106]. It is a graphical EDA tool that uses generator plugins to create the RTL design.

It is important to emphasize that interface standards describe only an IP's structure, interface and memory layout, but not the full functionality associated with each register. Therefore, they cannot completely replace the hardware description language. Leber provides a detailed description of an exemplary generator approach that produces the RTL structure of an efficient register interface implementations in [122]. The following describes a simplified RTL design as generated by most generators.

A register interface consists of basic sub-components such as the bus slave connected to the processor and memory controller via a standard bus such as AHB, APB, AXI or a customized bus. The bus interface generally provides decoder logic specific to the bus protocol. It processes the address, data, read/write and error/acknowledge signals and thus establishes a connection between the application logic and the register. The logic resolves the write and read request to an address location assigned to the IP address space. Write operations that match an address generate an internal write enable for one or more registers. A read operation, on the other hand, returns the register contents of the addressed register. With the previously introduced standards, it is possible to define individual properties for each register and each bit field, such as read/write permissions, virtual registers, and parity checks. Thus, a read or write transfer can also cause an error response or different behavior.

The available register interface generators mentioned before have different advantages. They differ in input/output formats, GUI support, bus protocols, underlying programming language and register properties. While the standard format of SystemRDL and IP-XACT is widely accepted in the industry, a particular generator has not yet gained acceptance. Design teams and companies often develop their own generators to create the HDL interface implementation. One reason is to avoid licensing costs and retain the ability to implement new features. Second, their designs would not differ from those of other vendors if they re-use common generators.

***Differentiation from related work**: The register interface generator used in this work follows the approach described in 3.2.2. The idea is that the register interface generator is integrated into the IP generation flow by using the same tooling. Thus, an instance of a hardware-software interface metamodel (RI-CIM) is automatically created when the IP-CIM is transformed into the IP-PIM. The RI-CIM can use the MDA generation flow to create the register interface. This way, an IP's complete hardware, including the IP core and the register interface, can be created.*

3.3.4 Hardware Abstraction Layer

The hardware abstraction layer is directly above the hardware register interface. The HAL introduces a unified interface with low-level access functions to the IP registers. A higher embedded software layer can call the HAL to read or write bit fields or entire registers. A specification defining the memory layout can be transformed into the HAL to provide a robust layer for the device driver to interact with the hardware.

Many tools provide a simple generator that uses SystemRDL, IP-XACT or other specifications to map the register layout into embedded software's header files. For instance, they translate the register layout into C structures or C macros [122]. However, such a header file misses the behavioral aspects and primarily describes the HAL structure or skeleton, while the implementation of the register accesses still needs to be done manually. A sequence of register operations can be combined into a single function call. When implementing such a function call, the relationships between the registers and bitfields must be known in detail. In addition to the dependencies, the characteristics of each register and bitfield must be analyzed to ensure a reliable and efficient function implementation. Especially for more extensive access sequences, manual source code can be either inefficient or faulty if one particular feature is not taken into account. A generator can automatically resolve the dependencies to build correct and optimized access sequences.

The work of [105] extends the format of IP-XACT with FW-relevant properties to specify so-called FW behaviors enabling the generation approach discussed so far. Similarly, the *SequenceEditor* of *Magillem* [15] and the *ISequenceSpec* of *DesignSpec* [26] allow developers to model register, and bitfield accesses sequences. Furthermore, the languages *HAIL* [195] and *Devil* [137] provide a domain-specific language that provides a high-level definition of the communication with a device and thus simplifies device access programming. However, all mentioned approaches describe hardware operations based on sequences. These can be transformed into a powerful HAL that allows direct access to registers on the fly without worrying about the underlying register layout.

***Differentiation from related work**: The hardware software interface metamodel in this thesis supports the construction of complex register access sequences. Furthermore, the generator can produce different HAL implementation variants, either memory-efficient or performance-optimized R31. In this work, the interaction between HW and FW is considered a key factor for the system's performance. Therefore, it offers an automatic optimization step that modifies the register bitfield mapping so that the sequences run as fast as possible without adding much hardware area and costs R30.*

3.3.5 Device Driver Layer

Above the HAL is the device driver layer, which provides hardware-related services. This layer is located below the application layer (optionally, a system can also contain an operating system that is located between the application and the driver). The driver layer abstracts the hardware completely into generic terms. For example, a UART driver has an API defined in terms of communication operations, such as receive and transmit functions. The description languages like HAIL [195] and DEVIL [137] specify low-level access functions hiding the low-level details of bit-level programming. They are limited to register accesses and do not offer all features needed to design the control and data flow of device driver programming.

DEVIL+ [232] is an extension of *DEVIL*, designed especially to describe the device's basic communication protocols. Wang et al. [218] extend *DEVIL* to synthesize more dedicated parts of a device driver automatically. It introduces a platform-independent specification that can be translated into a virtual environment and further mapped to a platform-specific driver implementation. A DSL for driver development is proposed through *NDL* [55] and *Laddie* [225]. They provide high-level device driver development constructs and use state machines to specify driver behavior. Both DSLs provide limited capabilities to describe the data- and control flow, covering only portions of the device driver core. Furthermore, they are based on a single level of abstraction and provide only C as a possible output language.

However, abstraction can be achieved through approaches other than modeling. For example, with "mbeddr" [216, 217], a proper language extension for C to develop abstract embedded software artifacts is offered. The extension includes modular constructs and notations that are integrated into the target C code. During generation, these constructs are mapped to C code. Others apply better programming methods to design embedded software, such as object-oriented programming and inheritance. Instead of programming in C, programming in high-level languages such as Java [161] or Python [206] is suggested. Such languages can simplify coding, but they are no substitute for automatic generation.

An automatic driver generation process must be highly dependent on the hardware specification. One way to achieve hardware dependency is to generate drivers from RTL test benches [38]. Another approach is to use a common specification of the system or the peripheral's hardware and software for driver generation. *Termite* [165] proposes an approach to driver synthesis based on a device and OS specification. The specifications define communication or "access protocols" as state machines used to create drivers that operate correctly and do not misconfigure the hardware. It can be applied as an extension of *DEVIL*, and other register access generation approaches. The *Me3D* [52] methodology addresses the problems faced by *Termite*, which models behavior using only state machines, resulting in an explosion of states and a large code size. *Me3D* takes several specifications, namely a model of the device properties, hardware specifications, an in-kernel interface specification, libraries and driver configuration parameters, to create a device driver.

Even closer to the hardware and even more versatile is the generation flow *DDGEN*, which is provided by Pendharkar and Kolathur [154]. It defines a domain-specific language called *DPS* (Device Programming Sequence) that captures the details of device attributes and behaviors in several device classes, e.g., interrupt specification and FIFO specification. A different solution is taken by Tanguy et al. [203], proposing a generation flow that includes more information about

the application to build drivers. Such an application-specific approach reduces the number of abstraction layers between the application and driver. In addition, it positively impacts performance and memory footprint since it generates only the necessary behavior.

Many other approaches claim correctness-by-construction but offer only a semi-automatic approach [7, 38, 109, 128, 232]. Manual post-processing is always required, which in turn can lead to errors in the design. For example, a template generator [7] that adapts to different embedded platforms is a typical use case for reducing coding effort, leaving the behavior coding to the designer. An exception is the tools [52, 154, 203], which claim to generate complete device driver code in C automatically. However, they omit an intermediate abstraction layer and generate the target code directly from the specification. This limits the generation capabilities and extensibility of the tool.

***Differentiation from related work**: A driver generation flow, such as the one envisioned in this work, must be able to create drivers that can be plugged directly into a compiler backend without the need for manual post-processing R5, R22. In addition, this thesis proposes more than one abstraction layer to enable platform-independent transformations and to be extensible in the future. In this way, the generation flow can be tuned to specific design metrics and generate C code and other target languages.*

3.3.6 Application

The application code contains no driver code but has access to the hardware through the driver layer interface, which hides the device's details from the application developer. A model-based approach for building applications is challenging since applications have a vast design space. Conversely, the driver's design space is constrained by the IP specifications. In general, all known approaches assist in modeling applications and systems but do not offer complete generation approaches, as this exceeds the complexity.

Panunzio and Vardanega [149, 150] mitigate the complexity by splitting the application into reusable blocks. It considers a component-based software engineering approach [88], where the application is created as a composition of application components interacting with each other. This keeps the application modular and well-organized. Other approaches derive application code from UML diagrams, e.g., activity diagrams [107].

A common approach of designers is to use graphical editing tools for UML to design applications. For example, *Papyrus* [121], *Sparx Systems Enterprise Architect* [11] or *VisualParadigm* [151] enabled a more general way of modeling software behavior, which turns UML behavior charts, e.g., sequence diagrams, activity diagrams or state machines, into software skeletons. In order to also derive the complete behavior, the designer must specify the target code in UML artifacts, e.g., state transitions and actions. Furthermore, the existing UML profiles are not tailored to the embedded domain.

MARTE (Modeling and Analysis of Real-Time Systems) [80, 134] defines UML profiles that capture detailed information about the application, the platform attributes and its architecture. Thus, it covers the whole design flow and provides a model that supports the HW/SW code design of complex embedded systems. The UML profile is suitable for modeling an application with structural and behavioral aspects. MARTE specifies the execution platform as a set of connected

resources, where each resource provides services to support the execution of the application [134].

MARTE can be used as a single-source specification to enable HW/SW code design activities for complex systems [89]. The approaches in [58, 60] use MARTE profiles as input models for the automatic generation of RTL code or software skeletons. [123] introduces a code generation strategy that transforms MARTE's application models into compilable C code. An approach for building *OpenCL* from MARTE is proposed in [162]. *gaspard2* [20] is a design environment that follows MDA to design data-intensive applications and hardware platforms from MARTE models. The use of MDA to design IPs and their application is further encouraged in [229]. The *MODES* framework [61] uses UML/MARTE for the functional specification and generation of the embedded application. *MODES* transforms UML models into internal models representing the application, capturing the functionality using processes communicating through ports and channels. The *ENOSYS* [43] project takes this one step further and presents a design flow that generates hardware and software and explores the design space. The core behavior is defined through activity diagrams and state machines, while an automatic partitioning step assigns the behavior patterns to software and hardware components. Similarly, [157] presents a methodology for automatically generating software and exploring different allocations of software components in real physical platforms.

***Differentiation from related work**: The embedded software metamodel presented in this work can be reused for application or library file generation in the same way it is used for device driver generation. An application can be defined through various specification metamodels, while generator templates handle the generation of the application, taking the driver interfaces into account. The embedded system flow for the entire stack, hardware, HW/SW interface, driver and application enables the automatic construction of embedded systems for FPGA, including synthesizable hardware and compileable software. Consequently, it enables automatic validation and design exploration R30 of various design decisions.*

3.4 Safety in embedded software

Safety mechanisms repeatedly appear in safety-critical systems, and their automatic generation can shorten the development time for such systems. The BMBF project *SAFE4I*[12] proposes ideas to accelerate the development of functionally safe software. *SAFE4I* proposes a strict separation between the design of the software functionality and the software safety measures.

Studies [201, 213] propose a metamodel with concepts and relationships for safety to facilitate safety compliance. It consists of entities and relationships that abstract concepts common to different safety standards. In [51, 119, 131], a UML profile for developing of safety-critical embedded systems compliant with the standard IEC 61508 [30] is proposed.

Some of the most applied safety patterns are documented in [63, 87, 115, 158]. Armoush et al. [12] propose a template representation of the safety patterns. A model representation of some selected safety design patterns can be found in [10]. The safety patterns contain all the required information to construct the safety mechanisms, enabling the automatic application of the safety measures through generators. Gleirscher and Kugele summarize the applied research on design

[12] https://www.edacentrum.de/safe4i/

patterns for ensuring system safety [81].

A generative approach that is not relying on MDE is provided in [209]. It suggests a DSL derived from AUTOSAR to formalize safety requirements and mechanisms and uses *mbeddr's* [216] functionality to customize and integrate safety mechanisms into the design. This approach makes integration into other frameworks difficult. Several other approaches use standardized modeling languages to design safety measures.

[155, 170] proposes a methodology for distributed run-time monitors based on UML and design-by-contract [62, 138]. The main idea is to extend the interfaces of software components with contract specifications (intended behavior) that have to be fulfilled at run-time. In [153], a methodology to define run-time monitors is presented. A generative solution for software-based memory protection is discussed in [39, 94, 152]. In general, many approaches focus on a specific safety measure but do not provide a generic flow that supports diverse concepts.

A versatile framework that provides a set of code generators for various safety measures is proposed by Huning et al.[93, 94, 95]. The framework supports the following categories of safety measures: (1) Object protection, such as *Cycling Redundancy Checksum* (CRC), *Triple Modular Redundancy* (TMR), *Range Checks* or time-based *Update Checks*. (2) Voting mechanisms, e.g., majority or plurality voting. (3) Time-based constraint monitors, e.g., watchdogs, to detect time-related errors. In addition, the framework is designed to generate both software- and hardware-implemented safety mechanisms and to enable the generation of error-handling procedures [93].

***Differentiation from related work**: In this thesis, the safety modeling is separated from the functional modeling R2. The safety pattern can optionally be extended on the functional-correct abstract embedded software model (FW-PIM). This framework is designed to simplify the extension with further safety measures R6. This work proposes a highly configurable time and control-flow-based safety patterns, which can be studied as design-by-example.*

Chapter 4

View Generator and Language Model

Code generation describes a mechanism that converts an abstract description (e.g., in UML) into a target language source code. A well-deployed code generator increases productivity since the generator is written with a one-time effort but can be reused indefinitely. It also increases the target code's quality: First, it reduces the risk of coding errors that are often introduced into even the most repetitive code when manually written. Second, it preserves the consistency of the code and its style between different projects and developers. Third, it increases consistency to abstract specification models.

Model-driven architecture's central concept is the separation between conceptual design and a concrete implementation and automation by several transformation steps. MDA conceives the code generation as the final model-to-text transformation step applied on the PSMs to complete the end-to-end generation. A PSM, defined on the platform-specific layer, describes the PIM extended by implementation artifacts tailored to the particular computing platform's properties (e.g., bit field or state machine schema). The presented framework defines a PSM as a language model of, e.g., C or `C++`. This model contains all implementation details of the target view to feed the code generator.

This chapter presents a novel approach that significantly simplifies code generation by moving from a direct generation of target code with print-like statements to the view model's assembling. Furthermore, it introduces a domain-independent mechanism to construct any view generator consisting of the platform-specific metamodel, its API and the model-to-text transformation template. The view generator's construction is triggered by a single source that captures the meta-syntax of a formal language[13], named as view language description (VLD). This approach simplifies the construction of new code generators while coping with code generation's two biggest challenges: maintenance and complexity.

4.1 View Generator

Each step in the design process of an embedded system can be subject to different views, e.g., VHDL and Verilog for the HW and TeX or VBA for the documentation. In the embedded software domain, common views are C, `C++` or Rust. Each of these views has a specific implementation style and is realized through its dedicated view generator. The principle of the view generator, shown

[13] A formal language encodes a grammar containing a specific set of production rules. It is, among others, conceived to define the syntactical aspects of a programming language.

in Figure 4.1, is to assemble a PSM of the target language. For this purpose, a platform-specific metamodel (PSMM) is introduced to capture the target view's meta-syntax. An instance of this metamodel derives part (skeleton) or all of the source code.

The view generator relies on the metamodeling environment, which automatically provides an API for each PSMM used to populate and read the PSM in a transformation script. This way, the designer can employ an IP-independent transformation in Section Design Pattern to map the PIM to the PSM. Furthermore, the view generator includes the Unparser that performs the model-to-text transformation.

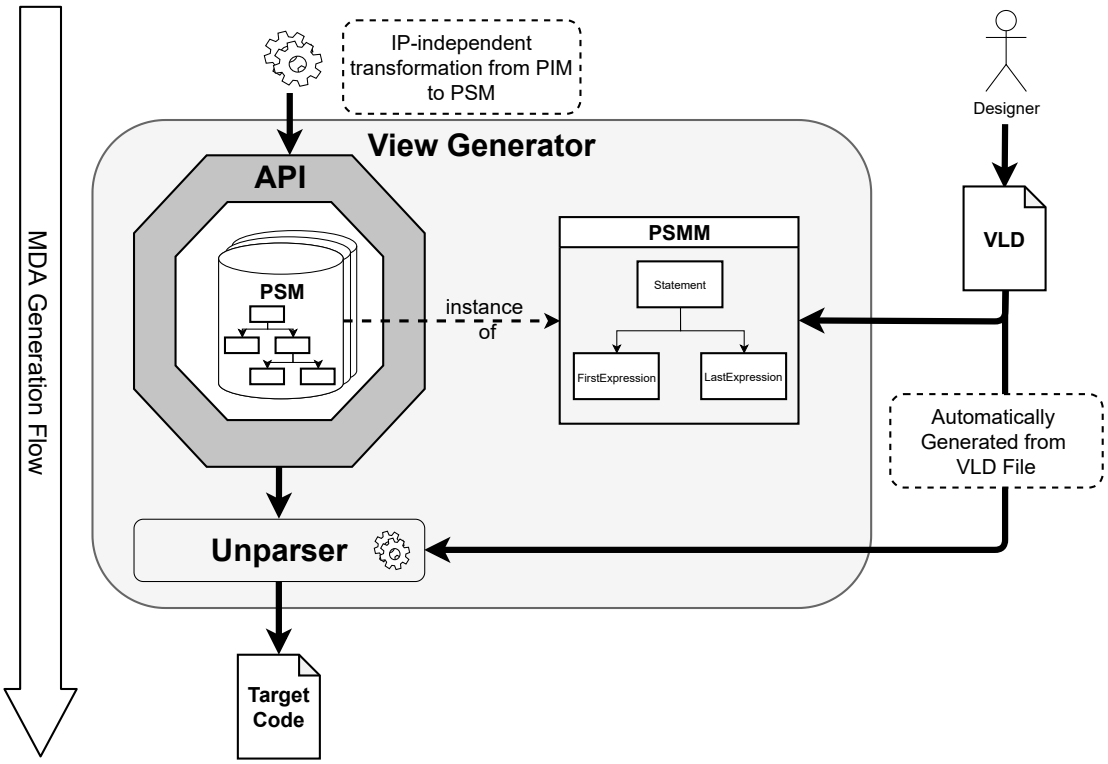


Figure 4.1: *Architecture of the view generator framework.*

4.1.1 Construction of a View Generator

Each view generator supports a different target view in the MDA framework. The effort to provide a view generator including all its components is low since it is automatically generated from a single source. The designer only needs to specify the view language description in a one-time effort. The individual components of the view generator are created without the necessity of manual post-processing. A meta-syntax notation, such as the VLD, defines a formal description of a formal language, emphasizing model constructs and targeting code formalities. A VLD defines the structure of the source code through a set of production rules. Accordingly, it constrains the hierarchical structure of syntactically correct views. In the generation of the view generator, the VLD is mapped into a metamodel that allows describing any valid abstract syntax tree[14] (AST).

Similarly, the framework generates the Unparser class, which contains methods to render each

[14] An abstract syntax tree represents the abstract syntactic structure of a language through constructs. However, it does not represent any inessential details of the language.

production rule. The methods of the class traverse a PSM recursively and output the source code for each production rule. Thereby, the translation appends the model with language and formatting details defined in the VLD.

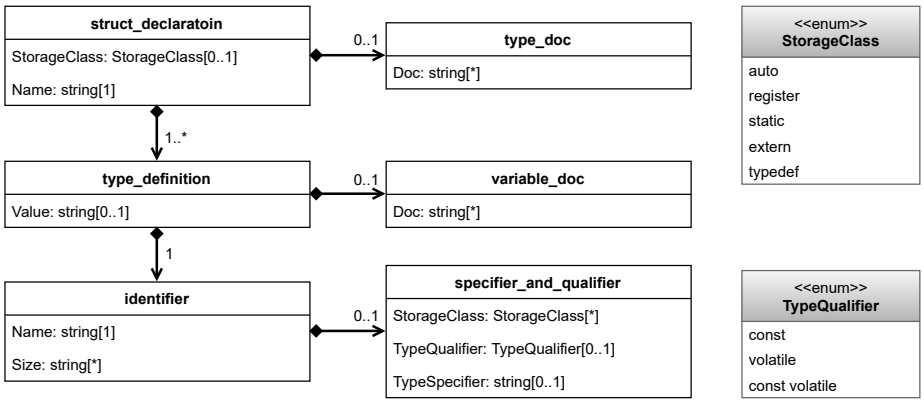


Figure 4.2: *Platform specific metamodel of a C-Structure.*

4.1.2 Abstract Language Model

In the generation flow's back-end, a designer can utilize different view generators. For the embedded software domain, a designer constructs a valid PSM that describes part or all the software architectural layer's target. Each node of the PSM denotes a construct occurring in the target code. For example, Figure 4.2 shows a part of the C view metamodel describing a C-Structure's essential constructs.

A *struct_ declaration* of the C-PSMM defines a data type with an ordered sequence of one or more *type_ definition*. A *type_ definition* specifies a list of structure members. Each contains an *identifier* given by a *Name* and *specifier_ and_ qualifier* which is optionally assigning a default *Value*. Furthermore, it can be specified as a static array member with a fixed *Size*. The designer can optionally document the structure, but also its members.

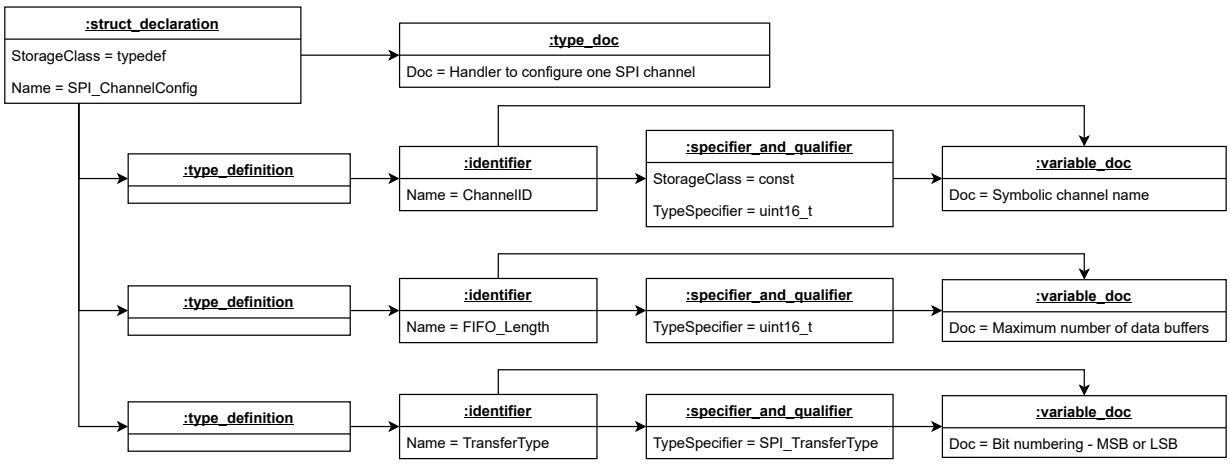


Figure 4.3: *Platform specific model of the SPI_ ChannelConfig handler.*

Such a C-PSMM can take different manifestations by being instantiated as a function of a more abstract model, the PIM. An instance of the *struct_ declaration* is illustrated in Figure 4.3.

It depicts the device handler of an SPI channel derived from the SPI's PIM, including the members *ChannelID*, *FIFO_Length* and *TransferType*. In this way, a designer can populate any handler structure and the complete AST of the target view.

4.1.3 Unparser

The PSM captures the essence of the view and neglects irrelevant elements (e.g., formatting constructs or terminal symbols) of the language. The Unparser disassembles the PSM's AST and extends the syntax constructs, e.g., formatting directives, to generate the target file. For this purpose, the Unparser provides a set of transformation rules, which are utilized to translate all AST nodes.

Algorithm 1: Method of the Unparser for rendering a *struct_declaration* in C.

```
1  self = struct_declaration object from the PSM instance
2  print_list = String list that is written to the output file
   /* Optional constructs of a C-Structure.                    */
3  if self.hastype_doc() then
4  |  print_list.extend(self.gettype_doc().unparse())
5  if self.hasStorageClass() then
6  |  print_list.append(self.getStorageClass() + " ")
7  print_list.append("struct " + self.getName() + "{\n")
8  print_list0 = []
   /* List containing any number of variable declarations.     */
9  for x0 ∈ self.gettype_definitions() do
10 |  print_list0.extend(x0.unparse())
11 |  print_list0.append("\n")
12 print_list.extend(cf.indent(print_list0))
13 print_list.append("}" + self.getName())
```

The Unparser's method for rendering the class or the production rule of struct_declaration is provided in Algorithm 1. The function utilizes the API to assemble the string list of a C-Structure depending on the PSM, which is finally output as a section of the target file. The feeding of this method with the PSM of the SPI_ChannelConfig results in the target code of Listing 4.1.

Listing 4.1: *Generated device handler for the SPI channel*

```
/**
 * Handler to configure SPI channels
*/
typedef SPI_ChannelConfig{
    const uint16_t ChannelID; /**< Symbolic channel name*/
    uint16_t FIFO_Length; /**< Maximum number of data buffers*/
    SPI_TransferType TransferType; /**< Bit numbering: MSB - LSB*/
}SPI_ChannelConfig;
```

For the generation, all members of this class are traversed in sequential order and combined with terminal symbols (e.g., ‘‘`struct`’’ in line 7) and formatting rules (e.g., `indent()` in line 12) to obtain a legal sequence. This process considers cardinalities. For example, the C structure

optionally includes a type documentation (lines 3-4) and a storage class (lines 5-6) along with a repeating list of structure declarations (lines 9-11).

4.2 View Language Description

The view generator framework's central format is the View Language Description, which follows the idea of Extended Backus–Naur Form (EBNF) [175]. A VLD can be called a meta language since it describes any formal language's syntax. It provides a grammar that is substantially a list of production rules that combines single constructs into significant structures [173, 222].

A VLD defines the grammar of a language and thus exclusively the syntactic validity and not semantic correctness. In the MDA approach, the semantic validity is to a large extent guaranteed by further constraints defined in the PIM-to-PSM transformation script. For example, it prevents a struct_declaration from having multiple structure declarations with the same name.

Table 4.1: *Symbol table for EBNF and VLD*

Symbol Usage	Symbol in EBNF	Symbol in VLD
Rule definition	=	=
Rule termination	;	;
Concatenation	,	,
Alternation	|	|
Terminal String	"..."	"..."
Repetition	{...}*	{...}
Repetition (at least one)	{...}+	+...+
Grouped Elements	(...)	(...)

4.2.1 EBNF Distinction

A production rule of the EBNF standard ISO/IEC 14977 [192] defines a series of symbols, either non-terminal production rules or terminal symbols. VLD utilizes many aspects of EBNF's syntax and notation, such as special symbols listed in Table 4.1. VLD adds new symbols and directives to this notation to deal with formatting, which is a missing functionality of EBNF. The motivation behind this extension is apparent when considering a subset of the C syntax in EBNF (grammar adapted from [111]):

```
struct-specifier =    "struct", <identifier>, "{", {struct-declaration}+, "}"
struct-declaration =    {specifier-qualifier}*, struct-declarator-list
struct-declarator-list =    struct-declarator | struct-declarator-list, ",", struct-declarator
struct-declarator =    declarator | declarator, ":", constant-expression
```

The EBNF description only reflects a valid agreement on symbol order leaving out formatting symbols. It, therefore, introduces abstraction since any number of whitespace, tab and newline characters can be inserted between symbols. In most cases, this does not affect syntactic correctness but keeps the stylistic formatting abstract. This freedom can lead to unpretty-printing and inconsistencies in the target code, prohibited in, e.g., [19, 98]. For example, an AST of the

struct-specifier does describe multiple views and might result in a one-liner or a well-formatted code, as shown in Listing 4.1.

4.2.2 VLD Notation

The VLD serves as the source for generating all components of the view generator: the PSMM, its intuitive API and the Unparser. These requirements demand extensions to the EBNF notation by additional syntax directives. The VLD for generating the components (PSMM in Figure 4.2 and *Unparser* in (Algorithm 1) of a C structure is shown in the VLD production rule struct_-declaration.

It should be noted that VLD has high granularity compared to other formal languages. EBNF specifies terminal symbols as a concatenation of alphanumerical characters. Instead, VLD defines terminal symbols as common Python types added to the PSMM. By default, a symbol is of type string. For assigning another type, the symbol is customized by a construct `<name:type>`. An example is the symbol `<first:expression>` in arithmetic_expression. This notation has two advantages: First, it restricts the type range and thus increases consistency. Second, it facilitates the coding of the PIM-PSM transformation since variables can be directly assigned between models. For defining and referencing a custom enumeration[15] in the PSMM, the designer adds `@` as a prefix to the symbol. For example, struct_declaration refers to an optional enumeration type `<@StorageClass>`.

The VLD describes the formatting of production rules very precisely and specifies spaces, tabs (`"\t"`) and line breaks (`"\n"`) as terminal strings in the production rules. Further, the VLD provides special formatting directives included in the syntax, leaving for a source code that follows any formatting convention.

One of these functions addresses a frequent problem of formal languages: index-based formatting for repeating symbols. In many views, the surrounding terminal strings of the first or last instance of a repetitive sequence differ. For example, the EBNF production rule struct-declarator-list recursively defines struct-declarators separated by a comma. It requires two hierarchical production rules that invoke a recursive sequence of symbols, always adding a comma except for the last one. This structure increases the effort to design the code generator and the PIM-PSM transformation.

Instead, VLD defines a special formatting construct and stores such a recursive coding pattern as a list in the model. Therefore, it reduces the total number of hierarchical production rules significantly since it can handle such a pattern in a single production rule. The designer only has to populate the list ignoring the index of the symbol. The Unparser then carries out different formatting of each list item. For specifying item-based formatting, the list (either `{...}` or `+...+`) is surrounded by the formatting construct `%...%`. Inside this notation, Python index operators can be used to establish index-based formatting rules. A production rule that applies this formatting construct is the variable_doc. In this example, all elements except the last one (`[0:-1]`) of `DoC` are suffixed by a newline (`"\n"`) operator.

Another issue of formal languages is the indentation and alignment of subsequent coding

[15]The UML specifies an enumeration type as a class with the stereotype «enum». The attributes of the UML class are the values of an enumeration type.

lines. For this reason, the VLD supports a modular approach to describe special functions to handle different formatting needs. Such a function is describing a post-processing routine that encloses part of the production rule with `$<function_name>()$(...)`. A frequently applied post-processor is the `$indent()(...)$`, which adds a tab character to each line generated by the enclosed symbols. This notation is utilized within the struct_declaration to prepend an indentation for the enclosed type_definition. In this example, the Unparser in Algorithm 1 calls the post-processor `print_list.extend(cf.indent(print_list0))` after resolving the enclosed part of the production rule.

Since VLD focuses on un-parsing, the VLD rules cannot automatically be used to build a parser. In particular, VLD does not care whether the rules form an ll(1) or ll(k) grammar or a context-specific grammar. This is because the PSM is constructed by model transformation and not by parsing.

4.3 C Code Syntax in VLD

A survey from *EE Times* and *Embedded* on embedded software in [71] indicates domination of C in the embedded systems programming domain in 2019. C is the preferred choice in 56% of the evaluated embedded projects as it neatly combines low-level functionality with modern programming conventions. The described MDA flow for firmware generation can utilize various view generators in the back-end. One of these view generators is the C generator.

This section introduces the C-VLD, which serves as the basis for constructing the C code generator. The C-VLD is derived from the MISRA C guideline [19] recommended for software development in embedded systems. MISRA C describes a subset of the ANSI-C language [99] which defines rules constraining the language to increase reliability[16]. The guidelines focus on coding practices that describe language usage but also formatting rules to improve readability. The C-VLD production rules are designed to support these guidelines and to simplify the PIM-PSM transformation. It provides consistent rules on the one hand and rules with appropriate granularity on the other hand.

4.3.1 MISRA C Compliance

The MDA framework ensures compliance with a subset of the MISRA C rules listed in Table 4.2. C-VLD handles formatting- and syntax-related rules, such as. MISRA D15.6, which obligates the use of curly braces for the body of if, for, or while statements:

```
// Non-Compliant Code
if (condition)
    execute_control_block();
```

```
// Compliant Code
if (condition)
{
    execute_control_block();
}
```

These formatting rules have been extended with additional custom coding style and formatting guidelines.

[16]Note that a formal language defines the language's constructs but not how the compiler interprets them. MISRA C improves reliability by avoiding error-prone language constructs as they might be interpreted differently from compiler to compiler.

- Indentation of control blocks initialized by loops or conditional statements.
- Surround expressions and operators with white spaces.
- Opening ({) and closing (}) braces grouping instructions shall be on their own line.
- Only one statement per line.
- No combination of a statement and a line comment // in one line.

Besides syntax, MISRA C also defines rules to ensure semantically clean code. These capture essential aspects of program semantics that cannot be considered within a formal language, such as the VLD. Most of these rules avoid programming errors that are not detected by the compiler but lead to misbehavior. An example of misbehavior or undefined behavior arises with: Implicit type conversion, values outside range, uninitialized variables or type mismatches.

Static code analysis is a common approach to ensure code compliance with most of the guidelines. There are different techniques available, such as AST walker analysis or data flow analysis, which all rely on examining the source code AST. Each existing tool applies different techniques to check MISRA conformance [73]. A tool such as PVS Studio [198] can be plugged into the framework to perform static code analysis on the generated code.

Moreover, the MDA framework provides a built-in automatic checker to ensure compliance with many of the MISRA C rules, listed in Table 4.2. The checker either forces compliance or throws a warning within the PIM-PSM transformation. Instead of analyzing the source code AST, the built-in checker processes the PIM, including the program control flow and the type system. The analysis covers the rules that deal with semantic errors inside a single target file or function, which requires no whole-program analysis[17]. Note that a program also references external libraries not included in the PIM. For example, one rule within the scope of the built-in checker is D5.3:

```
// Non-Compliant Code
uint16_t var_1;
if (condition){
    uint8_t var_1;
}
```

```
// Compliant Code
uint16_t var_1;
if (condition){
    uint8_t var_2;
}
```

4.4 C VLD Constructs

The type system of C language consists of expressions, statements and blocks [111]. An expression (`x + y`) is a construct that interconnects entities, e.g., variables and operators that yield a result value. An expression is a fragment of a statement that describes a single identifier or a nested expression tree. A statement can comprise multiple expressions to make up a complete execution unit or action (`a = b + c`). Further, a block or control block is a region of the program grouping multiple statements to a compound statement.

4.4.1 File structure

VLD specifies a syntactic file structure utilized to generate source (.c) and header (.h) files. The root node representing the entire file is the source_text. Note that each VLD must have exactly

[17] The built-in static code analysis excludes all pointer arithmetic rules that require pointer analysis.

Table 4.2: *The firmware generation framework handles MISRA C guidelines for using C languages in critical systems [19]. The rules are grouped into Mandatory(++), Required (+), Advisory (o). The highlighted rules are respected through the C-VLD; others are handled within the PIM-PSM transformation.*

Category	MISRA C rule	Reference Title
++	D2.2	There shall be no dead code.
+	D3.1	The character sequences /* and // shall not be used within a comment.
+	D4.3	Assembly language shall be encapsulated and isolated.
+	D4.5	Identifiers in the same name space with overlapping visibility should be typographically unambiguous.
+	D4.10	Prevent the contents of a header file being included more than once
++	D5.3	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.
+	D8.1	Types shall be explicitly specified.
+	D8.2	Function types shall be in prototype form with named parameters.
+	D8.3	All declarations of an object or function shall use the same names and type qualifiers.
+	D8.6	An identifier with external linkage shall have one external definition.
+	D8.12	Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique.
++	D9.1	The value of an object with automatic storage duration shall not be read before it has been set.
+	D9.2	The initializer of an aggregate shall be enclosed in braces.
+	D9.3	Arrays shall not be partially initialized.
+	D10.1	Operands shall not be of an inappropriate essential-type.
++	D10.4	Both operands of an arithmetic operator shall have the same type category.
o	D12.1	The precedence of operators within expressions should be made explicit.
+	D12.3	The comma operator should not be used.
+	D13.4	The result of an assignment operator should not be used.
++	D13.6	The operand of the sizeof operator shall not contain any expression which has potential side effects.
+	D14.1	A loop counter shall not have essentially floating type.
+	D14.2	A for loop shall be well-formed.
++	D15.1	The goto statement should not be used.
++	D15.5	A function should have a single point of exit at the end.
+	D15.6	The body of an iteration-statement or a selection-statement shall be a compound-statement
++	D17.2	Functions shall not call themselves, either directly or indirectly.
++	D17.4	The exit path from a function with non-void return type shall have an explicit return statement with an expression.
o	D19.2	The union keyword should not be used
o	D20.1	#include directives should only be preceded by preprocessor directives or comments
+	D20.2	The ', " or \characters and the /* , // , or non-standard character sequences shall not occur in a header file name in #include
+	D20.4	A macro shall not be defined with the same name as a keyword.
o	D20.5	#undef should not be used.
+	D20.6	Tokens that look like a preprocessing directive shall not occur within a macro argument.

one root node without any brunches on top of it. An ifndef_description wraps the document with preprocessor directives considering D4.10.

```
source_text =    +description+;
description =    normal_description | ifndef_description;
ifndef_description =    "#ifndef ", <Name>, "\n#define ", <Name>,
                        "\n\n", normal_description, "\n\n#endif";
normal_description =    "// AUTO GENERATED CODE // \n", [file_doc],
                        {preprocessor_statement, "\n"},"\n",
                        {type_statement, ";\n"},
                        {function_declaration, ";\n"},
                        {function_definition, ";\n"};
```

The file is organized into four sections. The file starts with a set of preprocessor_statements, initializing an ordered sequence of C-preprocessors (D20.1). The section is followed by a list of type_statements, which include type declarations and type definitions. Finally, the file first specifies a section with all function_declarations, followed by a section with all function_definitions.

Depending on the target file, the sections are assembled differently by the PIM-PSM transformation script. A header file collects all sections with declarations, such as macro definitions, type declarations, and function declarations. So, it omits the function_definition section. On the other hand, the source file includes all definitions and largely omits the sections of function_declaration and preprocessor_statements.

4.4.2 Data Types

The C-VLD provides terminology for declaring and defining identifiers in the program. In general, type declarations and definitions are intended to be within the type_statement section of the file, but can also be moved to any other scope. A declaration introduces an identifier in the compilation unit without allocating memory. It specifies the properties of the identifier and its storage type, which can be either a C-primitive data type or a custom type. A `typedef` declaration specifies a custom data type. According to D19.2, the VLD omits the `union` type and supports the following type_statements:

```
type_statement =    enum_declaration | struct_declaration | type_definition | array_definition;
```

The class type_definition performs a simple declaration or definition of a single identifier (D12.3). A declaration specifies a named identifier with an enumerated list of declaration_specifier. The class can also be utilized as a pointer or array declaration. In contrast, a definition allocates memory and instantiates a previously declared identifier according to its declaration. It assigns a value to the memory area that was reserved through the identifier. The declaration and definition can be split into two type_definitions, resulting in two lines of code, or it can be handled in a single type_definition respecting D8.1.

```
type_definition =   identifier, [ " = ", <Value>], "; ", [variable_doc];
identifier =   [specifier_and_qualfier, " "], <Name>, {"[", <Size>, "]"};
specifier_and_qualfier =   {<@StorageClass>, " "}, [<@TypeQualifier>, " "], [<TypeSpecifier>];
```

The struct_declaration introduces a `struct` type into the compilation unit, which comprises a list of various type_definitions. The properties of the struct_declaration have already been discussed in detail in the previous sections of this chapter. It is only used for declaring a structure with all its members. In contrast, the definition of a structure is carried out to other classes, as shown in the following coding snippet:

```
// Initialization with type_definition.
struct SPI_ChannelConfig spi_ch_1; /* Uninitialized */
struct SPI_ChannelConfig spi_ch_2 = temp_spi_ch; /* Reference members from existing object */
// Initialization with array_definition.
struct SPI_ChannelConfig spi_ch_3 = { 3, 32, MSB }; /* Initialization of contiguous memberss */
```

C handles structure and array definitions in the same way since both initialize a list of members. The MDA framework exploits this fact and specifies a uniform approach to construct both in a single production rule called array_definition. This production rule allows the composition of all kinds of nested structures or multidimensional arrays, respecting the rules of D9.2 and D9.3.

An enum_declaration introduces a new enum tag, describing a set of named integer constants. The VLD notation always starts with the constant value 0 and does not allow implicitly specified enumeration constants to prevent the issue mentioned in D8.12.

```
struct_declaration =   [type_doc], [<@StorageClass>, " "], "struct ", <Name>, "{\n",
                       $indent()$(+type_definition, "\n"+), "}", <Name>;

enum_declaration =   [type_doc], [<@StorageClass>, " "], "enum ", <Name>, "{ \n",
                     $indent()$(%+<Value>+ [0:-1]:",\n" [-1]:"\n}"%), <Name>;

array_definition =   identifier, " = { ", %{array} [0:-1]:", "%, " };", [variable_doc];
array =   new_array | value_array;
value_array =   %{expression} [0:-1]:", "%;
new_array =   "\n{ ", %{array} [0:-1]:" }, \n{ " [-1]:" }"% ;
```

It is important to note that the type_definition section of the PSM can instantiate type declarations and definitions in an arbitrary order maintained in the generated target code. A rigid mapping of the order derived from the PIM can cause forwarding declarations. Instead, a built-in mechanism in the PIM-PSM transformation checks and corrects the order by analyzing dependencies between declarations and definitions.

The function_declarations are dumped in a different file section. It specifies the name of the function, the expected identifier and the return type. Moreover, an addon can extend a function, allowing to specify special function attributes (`__attribute__(...)`) are handled by the compiler.

Each function_declaration has an associated function_definition with function body in another file section unless it is declared as an external function.

function_declaration = [function_doc, "\n"], [<@FunctionSpecifier>, " "], specifier_and_qualfier, " ", <Name>, "(", %{identifier} **[0:-1]:**", "%, ")", [" ", <Addon>];

4.4.3 Expressions

In a formal language, an expression is a fragment of code that evaluates to a return value. It may consist of a single entity (operand), such as an identifier, or a contiguous combination of such entities with C operators. The data type returned by an expression depends on the entities and the operator.

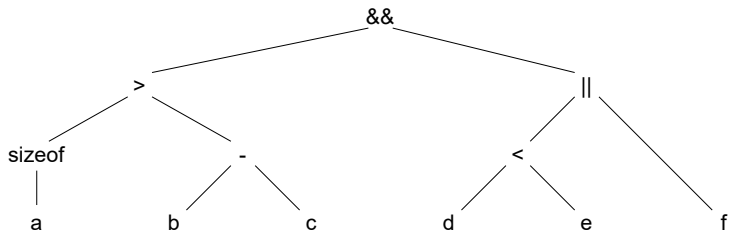


Figure 4.4: *Abstract binary expression tree of the PSM.*

Multiple sub-expressions can be combined to form a compound expression of certain execution order. This is achieved by the recursive structure of the expressions in the C-VLD, which contain either unary operator (e.g., logical NOT or `sizeof()`) or binary operators (e.g., arithmetic operation). In other words, the structure of expressions in the PSM resembles a binary expression tree. Consequently, the VLD does not support a conditional expression (`exp1 ? exp2 : exp3`) as it requires three expressions. Note that the function_expression is an exception to this rule.

In an expression tree, such as in Figure 4.4, each leaf is an identifier, while all internal nodes of the tree represent operators invoked by a sub-expression. In order to prevent the execution order, each expression defined as a production rule in the VLD is parenthesized. Accordingly, the generator cannot produce an ambiguous compound expression with more than two identifiers `x + y + z`. Instead, it always ensures an unambiguous order with parenthesis `(x + y ) + z`. So, the Unparser always generates target code that matches the execution order of the binary expression tree by applying the in-order traversal strategy (from left to right). For example, the introduced expression tree result: `((sizeof(a) > (b - c)) && ((d < e) || f))`

The C-VLD supports the following expressions.

expression = arithmetic_expression | rational_expression | logical_expression | prefix_expression | postfix_expression | function_expression | sizeof_expression | typecast_expression | pointer_expression | identifier | assembler_expression | plain_expression

The VLD introduces three types of expressions that can be utilized as binary expressions: arithmetic_expression, rational_expression, logical_expression. The structure of every binary

expression is similar since both contain two operands and a specific operator. This uniformity also simplifies the implementation of the PIM-PSM transformation.

The expressions differ only in the optionality of the first operand, which makes them also applicable as unary operators. A typical unary arithmetic operation is the inversion (-) of an expression. On the other hand, a unary logical operation is the NOT (!) operation that reverses the logical state of an operand.

```
arithmetic_expression =   "( ", [first:expression, " "], <@ArithmeticOperator>, " ", last:expression, " )";
rational_expression =   "( ", first:expression, " ", <@RationalOperator>, " ", last:expression, " )";
logical_expression =   "( ", [first:expression, " "], <@LogicalOperator>, " ", last:expression, " )";
```

Further, the C-VLD includes several other production rules that create unary expressions. The postfix_expression and prefix_expression increment or decrement an expression. Other unary expressions either extract information from the operand or cast the operand: sizeof_expression retrieves the size of an operand with the C built-in function `sizeof()`; typecast_expression explicitly converts the value of the operand from one data type to another; pointer_expression is used to either reference an operand (&) by taking its address or to dereference (*) an operand that maps a memory address.

```
postfix_expression =   "( ", expression, <@PostPreFix>, " )";
prefix_expression =   "( ", <@PostPreFix>, expression, " )";
```

```
sizeof_expression =   "sizeof( ", expression, " )";
typecast_expression =   "( ", <Type>, " )( ", expression, " )";
pointer_expression =   <@PointerOrAddr>, "( ", first:expression, " )";
```

The C-VLD provides unique expressions that may not follow binary or unary expressions and may not yield a value. That is why they can be considered as statements and expressions. For example, a function_expression realizes a function call containing any number of arguments, each represented by an expression, which can appear as a stand-alone expression without returning a value.

The assembler_expression is intended to express an obj_define_directive that describes an ordered list of inline assembler instructions. It is not supported by the PIM-PSM transformation to be called within a compound expression or compound statement D.4.3. A plain_expression is a backup solution if the designer requests a particular coding pattern that is not supported by the existing expressions. Both assembler_expression and plain_expression underlie no built-in checks and have to be applied with caution as they might lead to an erroneous behavior.

```
function_expression =   <Name>, "( ", %{expression} [0:-1]:", "%, " )";
assembler_expression =  "__asm__ __volatile__ (\\\n",
                        $indent()$( {"'", <Instruction>, "\\n' \\\n"},
                        $indent()$(": ", %{"=", <ROut>} [0:-1]:", "%, "\\\n"
                        $indent()$(": "), {<RIn>} [0:-1]:", "%, ")" );
plain_expression =  %{<Text>} [0:-1]:"\n"%;
```

4.4.4 Statements

Statements are the smallest standalone fragment of a C program that forms a complete unit of execution. It composes components like expressions to carry out a definite action. In the C-VLD, statements are divided into preprocessor_statements constituting the starting section of a file and basic statements that make up the body of a function.

The preprocessor_statement provides various directives invoked by the C preprocessor to transform the program before actual compilation. The most common preprocessor is the include_directive, which is used to include system-defined or user-defined header files. The preprocessor also handles conditional compilation through the ifdef_directive and ifndef_directive. The C-VLD provides only a limited set of conditional compilation statements since these are hardly needed in a model-driven generation framework. Such a framework already constructs the target code specification-dependently using conditional transformations. For describing macro definitions and extensions, the VLD provides object-like and function-like types of `#define` macros. The obj_define_directive describes an object-like macro that replaces an identifier with a constant expression in the compilation unit. The fct_define_directive can take additional arguments to make the macro function-like.

```
preprocessor_statement =  include_directive | obj_define_directive | fct_define_directive |
                          ifdef_directive | ifndefdirective | comment_line

    include_directive =   [type_doc], "#include ", <Name>;
    obj_define_directive =  [type_doc], "#define ", <Name>, " ",
                            $indent()$(expression );
    fct_define_directive =  [type_doc], "#define ", <Name>, "( ", %{identifier} [0:-1]:", "%,
                            " ) {\", $indent()$(+statement "; \\n"+), "}"
    ifndef_directive =  "#ifndef ", <Name>;
    ifdef_directive =   "#ifdef ", <Name>;
```

The function body is a control block made up of a compound statement which is initialized within the function_definition considering D13.4. A statement can be grouped into a simple statement, iteration statement, selection statement or jump statement. The simple statement, also named expression statement, consists of an expression followed by a semicolon, such as a function call using function_expression. The assignment_statement belongs to the simple statements which assign an expression to an identifier.

```
function_definition =   function_declaration, "{\n", $indent()$({statement, ";\n"}), "}";
statement =   for_statement | if_statement | while_statement | dowhile_statement |
              jump_statement | preprocessor_statement | expression | type_statement |
              assignment_statement;
```

```
assignment_statement =   <Name>, " ", <@AssignmentOperator>, " ", expression;
```

Another category are the iteration statements: `for, do...while and while`. The while_statement and the dowhile_statement have the same sub-components and are therefore initialized by the same PIM-PSM transformation. Both contain a compound statement and a control expression, which is evaluated in a different sequence. The for_statement inherits a compound statement but describes a loop proposed by D14.2 with a init-, a logic-, and an iterator-expression.

```
while_statement =   "while(", expression, "){\n", $indent()$({statement, ";\n"}), "}";
dowhile_statement =   "do {\n", $indent()$({statement, ";\n"}), "} while(", expression, ")";
for_statement =   "for (", init:expression, "; ", logic:expression, "; ", [ iterator:expression], "){\n",
                  $indent()$({statement, ";\n"}), "}";
```

The C-VLD covers one type of selection statement, the if_statement. The switch statement is not supported since its characteristics can also be realized by other coding patterns. The if_statement contains one if_expression, multiple elseif_expressions and an optional else_expression. Each of them also inherits a new control block as a compound statement. Additionally, the if_expression and elseif_expressions contain a control expression as well. The selection statements and the iteration statements can be considered as control statements that influence the program flow. They organize the program in a hierarchical control structure.

```
if_statement =   if_expression, {"\n", elseif_expression}, ["\n", else_expression];
if_expression =   "if (", expression, "){\n", $indent()$({statement, ";\n"}), "\n}";
elseif_expression =   "else if (", expression, "){\n", $indent()$({statement, ";\n"}), "\n}";
else_expression =   "else{", $indent()$({statement, ";\n"}), "\n}";
```

Furthermore, the C-VLD features jump_statements, limited to the three JumpOperator types `return`, `break` and `continue`. The `goto` is omitted as it belongs to the unstructured control flow statements and makes the code less readable and maintainable D15.1. For code documentation, the comment_line is included as a statement enabling to insert comments between statements considering D.3.1.

```
jump_statement =   <@JumpOperator>, [" ", expression];
```

```
comment_line =   "//", <Comment>;
```

4.4.5 Documentation

Besides simple comment_line, the C-VLD is designed to support Doxygen [210] as a standard tool for professional documentation. It includes specially formatted comments in the source code that Doxygen extracts to generate clear documentation. Such documentation shows a well-structured overview of the program and its constructs, e.g., (files, functions, variables). Accordingly, the VLD provides the following rules variable_doc, type_doc, function_doc, file_doc to document all program elements.

```
variable_doc = "/**< ", %{<Doc>} [0:-1]:"\n" %, "*/";
type_doc =    "\n**\n",
                " * ", <Doc>, "\n",
                "*/\n";
function_doc =   "/**\n",
                " * ", <Doc>, "\n",
                {" * @param ", <Parameter>, "\n"},
                [" * @return ", <Return>, "\n"],
                {" * @see ", <Reference>, "\n"},
                "**/";
file_doc =   "/************************************************/\n",
             " * @file ", <FileName>, "\n",
             [" * @author ", <Author>, "\n"],
             [" * @date ", <Date>, "\n"],
             [" * @version ", <Version>, "\n"],
             {" * ", <Doc>, "\n"},
             "*/\n";
```

Chapter 5

Register Interface and Hardware Abstraction Layer

This chapter describes the implementation and integration of the register interface[18] in the design process. The register interface is a reoccurring module in SoCs that enables the interaction between core and peripheral device (e.g., UART, SPI). It is required to operate the peripheral and to share device information such as operating modes, configuration parameters or data. This device information is stored in bit fields that are mapped to registers of an IP-dependent address space. A bit field is a group of bits that contain a specific information of the device. The allocation of bit fields in registers represent the memory layout of the device.

This chapter discusses the features of the register interface metamodel. An instance of the metamodel is automatically created while generating the hardware of the peripheral device. So, it defines bit fields depending on the specification model of the peripheral device. The register interface metamodel is used as a single source specification for the generation of two layers of the embedded stack. First, it generates the hardware of the register interface including the memory layout of the memory mapped IP-device, as well as its connection to the bus architecture. Second, the HAL is created from an instance of the register interface, which provides a higher-level layer of the interface for the the device driver. This layer specifies register accesses to the IP-device on the lowest embedded software architecture layer. The joint consideration of the software aspects as well as hardware aspects in the model avoid specification inconsistencies.

This chapter gives an overview of the metamodel and the generation process to construct the HW and the HAL. Furthermore, the chapter introduces an optimization step to optimize the memory layout and discusses the advantages over standard specifications, such as IP-XACT [4] or SystemRDL [97].

5.1 Abstract Model of the Register Interface

The abstract model of the Register Interface is shown in Figure 5.1. It is divided into two parts, one covering the requirements for the hardware, such as the memory layout or the bus interface. The other part is used to define access sequences to the registers of the peripheral device, relevant for the software generation. The root node of the metamodel is the *MetaRI* that contains various *Interfaces*. This feature allows the designer to connect the memory mapped device to multiple busses. The *Interface* describes the bus architecture connected to the peripheral. It specifies the bus protocol and the way the bus addresses the register of the peripheral.

[18] A distinction is made between the register interface and a HW/SW interface. A HW/SW interface describes only the bus interface and not the memory layout of a memory mapped device.

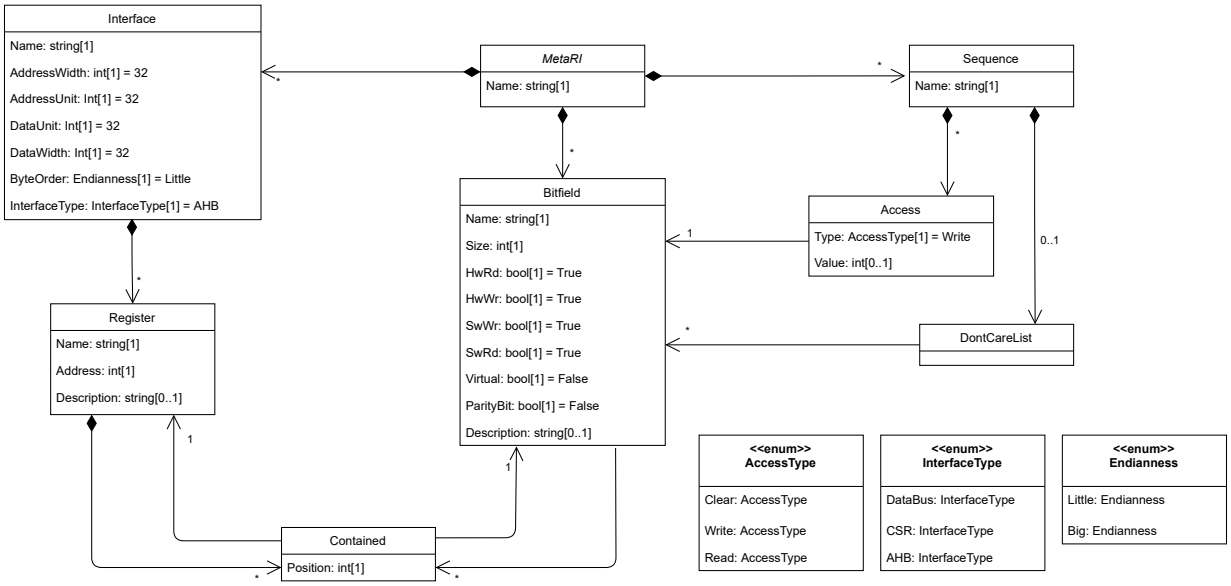


Figure 5.1: *The Register Interface Metamodel*

Three different bus protocols are supported by the metamodel through the *InterfaceType*. The *AHB* (Advanced High-performance Bus) specifies a common bus standard [187] from ARM providing a piplined protocol which is frequently used in industry. In contrast, the *SimpleBus* is an in-house protocol, presented in [224, 231], which is not supporting pipelined behavior. The CSR interface is a standard designed exclusively for RISC-V architectures to support atomic CSR instructions. In the CSR interface, the registers are defined as external CSR registers mapped to the addresses of the custom CSR address range[19].

The *ByteOrder* determines how the order of the byte array present on the data bus is resolved. Accordingly, an interface with a big-endian configuration stores the most significant byte of the byte array into the lowest memory address. In little-endian configuration the low-order byte is stored in the lowest memory address.

The *AddressWith* and *DataWidth* reflects the width of the connected address and data bus. The width of the data bus d_w determines the maximum number of bits that can be transferred with one bus access. The width of the address bus a_w limits the number of memory addresses of the peripheral to the maximum of 2^{a_w}. The smallest distance between two consecutive addresses in bits is configurable by *AddressUnit* a_u. In contrast to the *DataWidth*, the *DataUnit* d_u defines the smallest number of bits that can be transmitted with a single bus access. The ratio between d_w and a_u indicates the number of addressable units in the register and thus defines the supported store/load instructions for the peripheral. For example, $d_w = 32$ and $a_u = 8$ results in a register containing four 8-bit long addressable units. Each register is addressable through byte *(sb, lb)*, half-word *(sh, lh)* and word *(sw, lw)* store/read instructions. In comparison, an interface with $a_u = d_w$ supports only one access size. This gives the designer numerous options for optimization and a high degree of freedom[20].

[19] RISC-V maps the CSRs into a separate 12-bit address space for up to 4096 CSRs. In general, the different CSR lists are implemented as standard register file within the execution stage of the core. The CSR specification in [16] leaves an address range for custom uses.

[20] Notice: The RISC-V specification defines load and store instructions for access sizes from byte to double word.

The designer must consider the following definition while creating an instance of the register interface.

Definition 5.1. $d_w \in \{x = 2^{n+1} \mid n \in \mathbb{N}\}$

Definition 5.2. $a_w \in \{x = n + 1 \mid n \in \mathbb{N}\}$

Definition 5.3. $d_u, a_u \in \{x = 2^{n+1} \mid n \in \mathbb{N} \wedge n < \log_2(d_w)\}$

5.1.1 Flexible Memory Layout

The flexible design of the memory layout is a significant advantage of the metamodel compared to existing specification models described in 3.3.3. The *Interfaces*, *Registers* and *Bitfields* are decoupled from each other and can be arranged as preferred. An *Interface* contains multiple *Registers* with the size d_w, each addressable through a local *Address*[21]. Depending on the configuration of the interface, the register is divided into $\frac{d_w}{a_u}$ addressable units with adjacent addresses.

The *MetaRI* defines a set of $n_{BF} \in \mathbb{N}$ *Bitfields* $BF = \{BF_1, ..., BF_n\}$, without specifying the placement. In this way, *Bitfields* are not assigned explicitly to one of $n_R \in \mathbb{N}$ registers. The mapping of *Bitfields* to a certain *Position* of the *Registers* is achieved through the *Contained* object. The *Size* of each *Bitfield* determines the required space of adjacent bits in the *Register*. The total length of all included *Bitfields* must not exceed the size d_w. So, it shall be entirely within a *Register*. A valid mapping prohibits overlaps of *Bitfields*, but allows empty fields within the *Register*. Consequently, each register $r \in R = \{R_1, ..., R_{nR}\}$ allocates a multiset of n *Bitfields* $r = \{bf_1, bf_2, ..., bf_n\}$. This flexibility gives the designer control over many design choices.

However, the following design rules must be maintained.

Definition 5.4. $\forall bf \in BF : \exists r \in R \; with \; bf \in r$

The arrangement of the memory layout strongly influences the resulting hardware (see Section 5.2.1) and software design (see Section 5.2.2). A compact mapping is less performant but requires a smaller logic area, while a loose mapping is faster and bigger in logic area. This is the trade-off a developer must consider when defining a memory mapping.

5.1.2 Bitfield configurations

A designer configures each *Bitfield* in advance, either manually, or by a default configuration. The placement is initially not considered since it is left to the generator to decide on the final implementation, which gives freedom for optimizations. A chosen configuration for a *Bitfield* applies to all position where it is finally mapped.

Each *Bitfield* supports two types of access privileges. First, a read (*SwRd*) or write (*SwWr*) access via the bus. Without those, no software access from the HAL is possible. Second, it defines the access rights through the IP core (*HwWr* and *HwRd*) where the register interface is placed. A $bf \in BF$ is specified by $bf_{Sw,Hw,Sr,Hr}$ and satisfies the following definition.

For interfaces that allow further accesses, e.g. nibble or quadruple word, the instruction set must be extended by customer-specific load/store instructions.

[21] The physical address is mapped to the local address of the peripheral device in a separate slave interface that is connected in series.

Definition 5.5. $\forall bf_{Sw,Hw,Sr,Hr} \in BF : (Sw \vee Hw) \wedge (Sr \vee Hr)$

The *Virtual* attribute determines the availability of the storage elements that represent the *Bitfield*. A *Virtual Bitfield* indicates that the storage element is not created within the register interface, but signals for addressing the *Bitfield* are forwarded to the core. This feature enables individual modification of the read and write path of *Bitfields*.

Another feature is the *ParityBit*, which implements a safety measure by assigning a parity bit to the *Bitfield*. For overwriting a *Bitfield* successfully, the parity bit must be activated during the write operation. Accordingly, the size of the register is extended by one bit, which must be considered in the mapping.

5.1.3 Hardware Access Sequences

Reducing the number of bus accesses increases performance. This is achieved by combining hardware accesses into *Sequences* that are provided for the upper embedded software layer. A *Sequence* defines a combination of *Accesses* of different *Types*, e.g., *Write*, *Read* or *Clear*, that should be executed simultaneously in a single function. Each access in the *Sequence* references a *Bitfield* independent of its location. A *Write* access optionally defines a constant *Value* or a function argument that is written to the memory location. A sequence s is defined as:

Definition 5.6. $s = \bigcup \{(bf, a) \mid bf \in BF \wedge a \in \{Clear, Write, Read\}\}$

In addition, a designer can define a *DontCareList* for each *Sequence*, which references *Bitfields* that may be modified during the access sequence without affecting the behavior. This can speed up *Sequences*, since otherwise *Bitfields* of the same accessed address must be cached to prevent them from being modified.

5.2 Register Interface and HAL Generation

The instantiation of the register interface is accomplished within the IP generation, as shown in Figure 5.2. First, a default configuration of the CIM of the register interface is instantiated depending on the chosen specifications. The required bit fields are derived from the IP specification, while the bus information is extracted from the system specification[22]. For example, the CIM of a communication device defines the availability of a FIFO as data buffer. This results in the instantiation of bit fields such as FIFO_EMPTY, FIFO_FULL that would not otherwise occur. The generator-backend builds the HW from the CIM using MetaRTL. A parallel HAL generator constructs the access sequences and turns the CIM directly into a PSM.

Within the generation flow, the developer can manually refine the default memory layout of the IP or use an optimizer for automatic refinement. The optimizer determines a mapping with the best trade-off between performance, memory footprint and logic area, as shown in [178, 181, 182].

5.2.1 Register Interface Hardware

The generated SoC has a memory-mapped I/O architecture that uses a common bus for multiple peripherals, including the data memory. The applied busses are CPU-external systems that map

[22] An IP that is created without reference to a system specification assumes a standard bus.

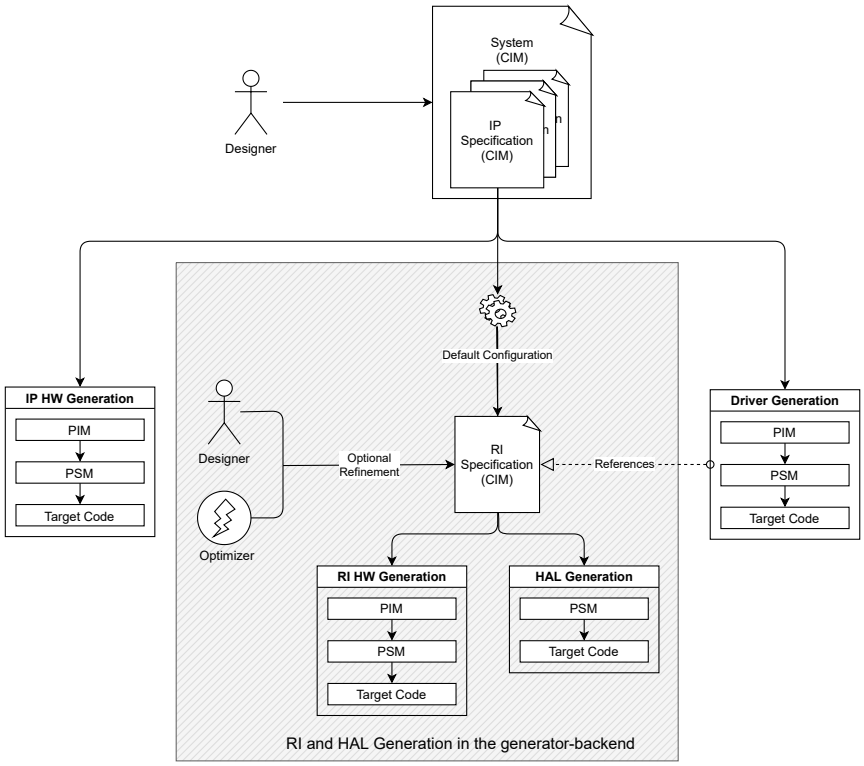


Figure 5.2: *The RI generation flow embedded in the generator-backend of the IP generation.*

memory or peripheral accesses. The number of bus architectures connected to the peripheral is unlimited. The intended design of different system bus architecture concepts is shown in Figure 5.3. This example illustrates a register interface connected to two bus systems (CSR and AHB). Each system bus consists of a signal bundle, which is composed as follows.

AHB: The signal *addr* carries an unidirectional signal bundle from master to slave specifiying the address of the addressed unit. The processor indicates with *addr_size* the size of the access (e.g., byte, half-word, word) that range between the smallest possible addressable unit a_u and the register size d_w. The size and existence of the signal *addr_size* depends on following definition:

Definition 5.7. $length(addr\_size) = \begin{cases} not\ generated, & if\ d_w = a_u \\ ceil[\log_2(\log_2(d_w) - log_2(a_u) + 1)], & otherwise \end{cases}$

The value $addr\_size = 0$ determines the biggest possible access. Each increment of the value halves the access size. The Table 5.1 lists the access options for a register interface configuration with $d_w = 32$ and $a_u = 8$.

The activation of a read or write access is triggered by the master, which enables the control signals *read_en* and *write_en*. The two data bus bundles *data_in* and *data_out* transfer the data between master and slave for the initiated operation.

CSR: The CSR bus is a specific bus for the RISC-V domain. A register interface of *InterfaceType* CSR, defines registers of the IP as CPU-external CSRs. This allows faster accesses through

Table 5.1: *All access combinations of a register interface with the configuration $d_w = 32$ and $a_u = 8$. A register within the interface has four addressable units (Bytes).*

All possible accesses	Access Size (*addr_size*)	Base Address (*addr*)	Access Mask
First Byte	0b10	0x0	0xFF000000
Second Byte	0b10	0x1	0x00FF0000
Third Byte	0b10	0x2	0x0000FF00
Fourth Byte	0b10	0x3	0x000000FF
First HW	0b01	0x0	0xFFFF0000
Second HW	0b01	0x2	0x0000FFFF
Word	0b00	0x0	0xFFFFFFFF

atomic CSR instructions. The RISC-V specification limits the address range[23] designated for custom use. Similar to the AHB, the CSR is addressed via the *addr* signal and the data is transmitted via *data_in* and *data_out*.

All CSR instructions atomically read-modify-write a single CSR. The two-bit signal *mode* references the type of operation, as shown in Table 5.2. Compared to the AHB, the CSR interface offers bit field-accurate accesses through mask-based operations. For a *CSRC* or *CSRS* operation, a mask applied on *data_in* defines all bit fields to be cleared or set. The bit-mask must match the mask of the bit field. In contrast, a *CSRW* instruction overwrites the entire register with the applied *data_in*. A CSR read instruction is encoded as a *CSRRS* instruction that transmits an empty mask to avoid modification of bit fields.

Furthermore, each CSR interface provides an unidirectional *error* signal that reports illegal accesses to the master. There are two reasons for this: The address is invalid; The applied mask is not matching a bit field.

Table 5.2: *The mode signal determines the CSR instruction. The bus applies the same mode for the immediate instructions, as well as the pseudo instructions.*

Access Type	*mode* signal	CSR Instruction
No Operation	0b00	-
[Read]-Modify-Write	0b01	CSRRW, CSRRWI, CSRW, CSRWI
[Read]-Modify-Set	0b10	CSRRS, CSRRSI, CSRS, CSRSI
[Read]-Modify-Clear	0b11	CSRRC, CSRRCI, CSRC, CSRCI

The hardware of the interface is defined with three sub-components, as shown in Figure 5.3. The decoder control logic processes the bus accesses from the master to the IP registers. It controls the relevant write and read control logic of the addressed register. These in turn activate single or multiple bit fields (in HW: flip flop arrays) within the register. Since the same bit fields can be located in different registers, they can be activated by different register control blocks. The two control logic blocks, write and read block, process the incoming and outgoing data.

The following applies to the activated bit fields during write access via the bus: The bit fields are within the addressed unit and are writable by software (*SwWr=True*); The parity bit

[23] The CSR Listing of RISC-V specification in [16] defines 192 addresses for custom read/write use in machine mode. The address range is between 0x7C0-0x7FF and 0xBC0-0xBFF

is set in *data_ in* when the bit field contains a parity bit. The peripheral logic, however, can also directly manipulate and read bit fields. For simultaneous access, the bus access is prioritized over peripheral access.

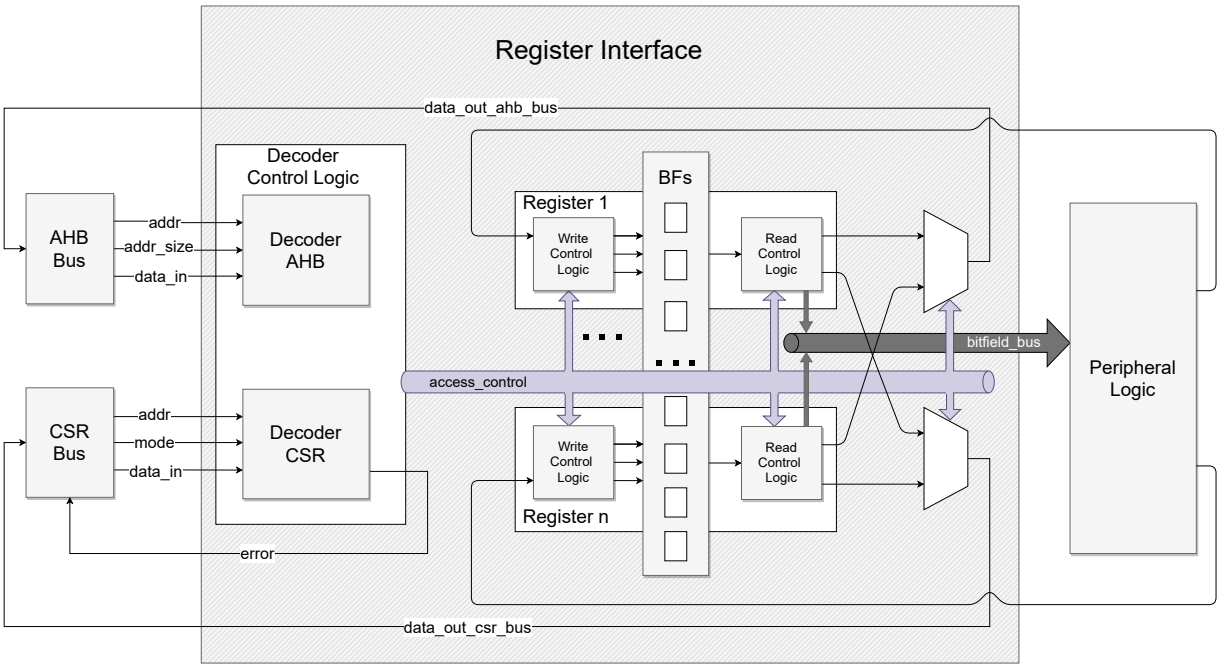


Figure 5.3: *Circuit block diagram of the register interface in the peripheral connected to two bus architectures (AHB and CSR).*

5.2.2 Hardware Abstraction Layer

The usability of the peripheral devices requires that the internal registers of the device can be read and written. The HAL is considered as an interface to the hardware that operates at the register level. In terms of software architecture layers, it acts as the intermediate layer between the register interface and the driver layer. The HAL maps device memory into address spaces and provides methods for the higher FW layers to perform operations on the registers. These methods are primarily bit manipulations on the peripheral registers of the interface.

The interaction between software and hardware can be achieved by different memory mapping techniques. For example, a memory efficient approach of dealing with bit-wise data are the `C/C++'s` bit field structs. However, this approach leaves a lot of responsibility to the compiler. A safer way to deal with bit fields is to employ shift and mask operations. Other approaches provide a generic HAL using customized data structures to describe the memory mapping. In industry, generating memory maps is attractive as it saves the effort of writing repetitive structures while reducing associated errors.

The framework includes the HAL generator, as shown in the RI flow of Figure 5.2. The HAL generator transforms the RI CIM directly into the PSM. The PIM is omitted since the level of abstraction between RI CIM and the target code is low. Compared to existing frameworks of 3.3.3, the generator allows to create different implementations of the mapping. Thus, the designer has the power to make different design decisions on the HAL layout. This allows the designer

to adapt the variant according to the application's demands affecting: code size, performance, portability. Two implementation variants are introduced in the following.

5.2.2.1 Implementation Variants

The generation of the HAL as a software architecture layer relies on the principal of R2 (Separation of Concerns). The HAL is constructed independently of higher software levels using a separate generator. The central FGC model provides individual control of the HAL generator for each peripheral device. Thus, the configurations of the HAL respectively the implementation variant can be adapted to the requirements of each individual system component. This impacts the interface signatures.

The device driver layer describes these hardware accesses as abstract entities in the PIM model. During transformation from PIM to PSM, these are replaced by the concrete signatures of the interface. Thus, the transformation from CIM to PIM solely focuses on the functionality of the peripheral driver without considering low-level properties (e.g., memory mappings and bit field configurations). This simplifies the implementation of the device driver generators significantly, considering the high proportion of hardware interaction. The following paragraphs present two implementation variants.

5.2.2.1.1 Generic Bitfield Structures A portable implementation variant of the HAL is generated using custom bit field structures. The structure's definition, represented in Listing 5.1, maps the position and length of each bit field within a register. The mask of each bit field can be derived from the structure itself. This makes the HAL highly generic.

Listing 5.1: *Type definition of the generic bit field structure.*

```
typedef struct BitField{
    uint8_t pos; /** Starting Position of the Bitfield */
    uint8_t length; /** Length of the Bitfield */
    uint32_t volatile* const Register; /** Pointer to the Register */
} BitField;
```

Bit field accesses can thus be realised by generic methods that reference the bit field structure for clearing, reading or writing. The function to set a single bit field is shown in Listing 5.2. The applicability of this implementation variant is limited and requires the following: Bit fields have to be configured without parity bits: For all $bf \in BF$ has to apply $Sw \wedge !Sr = 0$. If this does not apply, the generator throws a warning about potential misbehaviour. This is due to the fact that the structures and the generic functions only implement read-modify-write (RMW) accesses, ignoring parity bits. However, bit fields that do not meet the requirements need individual access strategies (e.g., with shadow variables).

Listing 5.2: *Generic write function for BitfieldStructs.*

```
static void bitfield_set(const BitField *bf, uint32_t val){
    uint32_t msk= BIT_MASK(bf->length);
    *(bf->Register)=((*(bf->Register) & ~(msk<<bf->pos)) | (val & msk)<<bf->pos);
}
```

The generic HAL reduces the memory footprint to a minimum[24] and defines a portable interface layer. However, a generic implementation lacks flexibility and leads to the following disadvantages. The performance is lower since each access is performed as a costly RMW operation. Furthermore, the implementation does not support all metamodel attributes. The biggest disadvantage of the approach is that it it fails to support bit field access sequences. Accordingly, there is also no possibility of optimisation.

5.2.2.1.2 Specific Inline Accesses This implementation variant is a counter-design to the previous bit field structures. Bit field accesses are individually designed and optimised in terms of their memory position and configuration. Depending on this, bit field actions are executed either by direct operation or through time-consuming RMW operations. The generation also includes shadow variables in the function if required. Thus, each generated access function has different costs.

The generated access functions are mapped into a function table, also called a function pointer array. The structure of the function table is thus dependent on the generated access functions and indirectly on the IP specification. A structure of the function table of the UART is shown in Listing 5.3. Each channel of the device[25] defines its individual entity of the function table.

Multiple entities are mapped into an array of function tables, which has several benefits. First, it enables the implementation of loop-based or iterative accesses to channels through indices, as required in initialisation and configuration functions, for example. Second, it increases portability, simplifying the implementation of design patterns on higher software layers. However, the program size also increases slightly and de-referencing requires extra CPU clock cycles.

Listing 5.3: *Function pointer array of the UART HAL.*

```
typedef struct UART_HAL_Config{
    void (*TX_ENABLE_UART_WRITE) (bool TX_enable_UART);
    bool (*TX_ENABLE_UART_READ) (void);
    void (*TX_DATA_UART_WRITE) (uint16_t TX_data_UART);
    void (*SEND_UART_WRITE) (bool send_UART);
    bool (*TX_FULL_UART_READ) (void);
    bool (*TX_EMPTY_UART_READ) (void);
    ...
}UART_HAL_Config;
```

5.2.3 Access Optimization

Hardware accesses comprise a major part of the overall behaviour. At the same time, they are a main cause of misbehaviour. Following the approach of correctness-by-construction, the HAL generator reduces error sources and improves code efficiency. Especially, when considering complex access combinations (*Sequences*) as introduced in Section 5.1.3.

Algorithm 2 outlines the control flow of the transformation for determining the best access function for a given sequence. It maximises performance by minimising the required number of

[24] The memory footprint is only reduced if the compiler leaves the function call for bit field access uninlined. Otherwise, the advantage of the generic HAL is lost due to its function overhead.

[25] Each IP device features one or more modules. A module is also considered as a channel.

Algorithm 2: Control flow that determines the generation of hardware accesses from a given sequence.

```
1  bf_active = Set of bit fields to be modified in the sequence s.
2  r_active = Minimal set of addressable units containing all bf_active
3  for bf_a ∈ bf_active do
4      if bf_a with Parity = True then
5          Set parity bit of bit field
6      if bf_a with SwRd = False then
7          Write value also to Shadow Variable
8  for r_i ∈ r_active do
9      bf_passive = bf ∈ r_i ∧ bf with Sw = True ∧ Parity = False
10     if bf_passive ≠ ∅ then
11         Direct access to bit fields;
12     else
13         for bf_p ∈ bf_passive do
14             if bf_p has Sr = False then
15                 Write back Shadow Variable
16             else
17                 RMW: Buffer current bit field variable
```

bus accesses and instructions for each sequence individually. The algorithm takes into account the properties of bit fields and the registers to determine the best transformation. Consequently, the generator realizes the sequence using one or more constructs of the following implementations:

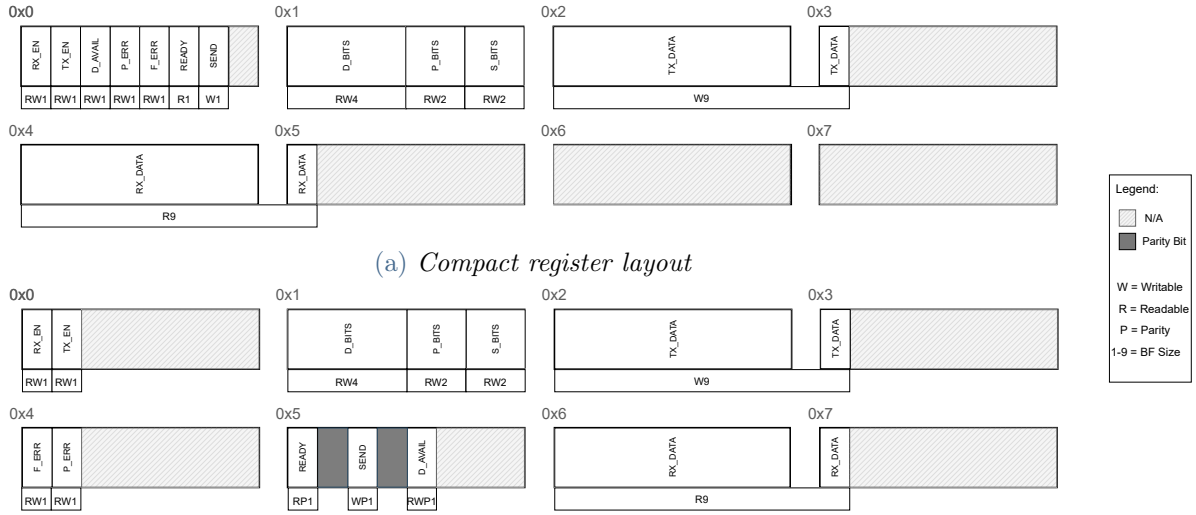
Direct Modification: The value at the data input is written directly into the addressable unit. This takes a single bus access. Direct modification is only permitted when all bit fields in the unit are either modified or protected.

Read-Modify-Write: For an RMW access, the addressable unit is cached, modified and written back. This function requires two bus accesses, but ensures that the bit fields within the addressable unit retain the old value. This sequence is required unless all bit fields are modified.

Shadow Variables: Shadow variables store the value of bit fields that are not readable via the bus as a copy in the RAM. The value of the HW bit field and the shadow variable must be consistent at all times. This must be considered for direct modifications, as well as for RMW. This implementation affects memory costs and performance (time for loading shadow variable).

According to these generator constructs, different memory layouts lead to different access implementations. For example, two different memory layouts of the UART device, as shown in Figure 5.4, affect the application's efficiency greatly. An initialisation sequence applied to the UART device is one of many sequences of the IP. It includes a single write access to *TX_EN* and *RX_EN*. The generator transforms the initialization sequence in dependence on the memory layout into different target code implementations, as shown in Listing 5.4. Processing the code of the compact layout takes 17 clock cycles, while the code of the optimised layout requires less than half of that (7 clock cycles)[26].

[26]The results were accomplished with GCC RISC-V 9.2.0 and optimisation -01. The number of cycles is derived



(a) *Compact register layout*

(b) *Optimized register layout*

Figure 5.4: *Two alternative memory allocations of a UART instance composed of a transmitter and receiver module.*

Listing 5.4: *Implementation of the UART initialization access sequence for different memory layouts.*

```
#define UART_CTRL_8_0 *(volatile uint8_t*) (0x0 + _BASE_ADDR_)
// Compact mapping of Figure 5.4 a): Shadow variables and RMW access
void enable_uart_rx_tx(bool rx_en, bool tx_en ){
    uint8_t tmp = UART_CTRL_8_0 & ~(TX_EN_8MSK | RX_EN_8MSK);
    tmp |= ((rx_en << RX_EN_8POS) & RX_EN_8MSK) | ((tx_en << TX_EN_8POS) & TX_EN_8MSK);
    UART_CTRL_8_0 = tmp | ((SEND_SHADOW << SEND_8POS) & SEND_8MSK);
}
// b) Optimized mapping of Figure 5.4 b): Direct access to register
void enable_uart_rx_tx(bool rx_en, bool tx_en ){
    UART_CTRL_8_0 = ((rx_en << RX_EN_8POS) & RX_EN_8MSK) | ((tx_en << TX_EN_8POS) & TX_EN_8MSK);
}
```

Similarly, performance also depends on the transfer sizes supported by the interface. By limiting the transfer size to 32-bit accesses ($a_u = 32$), the performance gain for the initialization sequence of the optimised memory layout decreases. Since both memory layouts now have to perform a time-consuming RMW operation to access *TX_EN* and *RX_EN*.

For further performance optimisation, the inclusion of parity bits is beneficial. For example, to signal the UART device that the received data has been read, a read-acknowledge sequence is defined. This sequence consist of a write access to *D_AVAIL* and clearing of the error bit fields (P_ERR and *F_ERR*). For the optimised memory layout of the Figure 5.4b the following applies: the generator ignores the bit field *SEND* since it is protected through a parity bit. The implementation of this access is illustrated in Listing 5.5.

from the number of instructions. The RMW operation of Listing 5.4 requires an additional write-back cycle.

Listing 5.5: *Implementation of a write access to the D_AVAIL bit field and clearing of P_ERR and F_ERR.*

```
#define UART_CTRL2_8_0 *(volatile uint8_t*) (0x4 + _BASE_ADDR_)
// b) Optimized mapping: Faster accesses through parity bits.
void enable_send_(bool d_avail){
    UART_CTRL2_16_0 = ((d_avail << D_AVAIL_16POS) & D_AVAIL_16MSK) | D_AVAIL_PARITY;
}
```

Chapter 6

Embedded Software Modeling

The generation of embedded software is organized in hierarchical subsystems of different IP components. In addition, each IP component is divided into several software architecture layers. The previous chapter introduced the generator approach for the lowest layer consisting of the Hardware Abstraction Layer. The generation followed a straightforward transformation from a register interface CIM to its PSM. Above the HAL resides the device driver layer, which provides hardware-related (respectively IP CIM related) functions. It is designed to enable interaction with the hardware device, using the HAL as a communication interface layer. Compared to the HAL, a device driver contains more complex elements to describe its control flow. The device driver layer also uses the three-layer MDA approach to create the behavior and structures of the firmware.

This chapter introduces an intermediate model, the FW PIM, to formally specify the structure and behavior of embedded software. It can also be perceived as an abstract firmware model equivalent to the HW PIM in the hardware generation flow. It is defined by a metamodel and can capture all relevant aspects of a particular subsystem to describe its device driver. However, it omits implementation and platform details, such as hardware access implementations, driver design and architecture, as MDA prescribes. The core structure of this embedded software model resembles that of a UML 2.0 activity diagram [164]. However, it adds the following aspects:

- More explicitly modeled actions (e.g., flow monitoring nodes to describe a checksum-based control-flow monitoring system) adapted to the embedded software domain.
- A firmware-specific type system (e.g., bit fields, device handlers).
- Combination of configuration attributes and behavior nodes in one model.
- Unified structure that simplifies the implementation of transformation scripts.

A designer can instantiate a FW PIM to create any embedded software subsystem, such as an application, device driver, or external library. The automated embedded software flow is divided in two generator stages: An IP-specific frontend and a generic, configurable backend. A designer defines a transformation for a particular subsystem (IP) in the frontend, which maps one or more CIMs[27] to one FW PIM. In the generator backend, the designer can apply customizable

[27]The designer may also choose to write a generator without CIM dependency. The generation results in a constant PIM, further translated into different PSMs.

IP-independent transformations to realize different design decisions (e.g., safety mechanisms, architecture and design styles). Both stages are not subject to fix transformation rules. Instead, they are controlled by a generator specification model customized by the designer.

6.1 Embedded Software Generation Flow

An embedded system comprises multiple IP devices, each specified by a configurable CIM that includes elementary features of the IP relevant for HW and FW development. Initially, a designer only chooses the scope of features by configuring these CIMs. Based on the selected specifications, the holistic embedded system generation process is performed. As shown in Figure 6.1, the embedded design flow uses the IP specification (CIM) as a starting point to drive the different generator flows. Thus, the entire flow is mastered by one entity, as described in the practice of a single source of truth (R1 (Single-Source Principle)). The embedded software generator is one of these generators that constructs the device driver for the particular IP design. Two advantages arise for the embedded software from the single-source design principle: First, it can be exploited to generate device drivers adapted or optimized to the HW. Second, it reduces the risk of conflicting duplicate information, ensuring the consistent and design-correct generation of hardware and firmware.

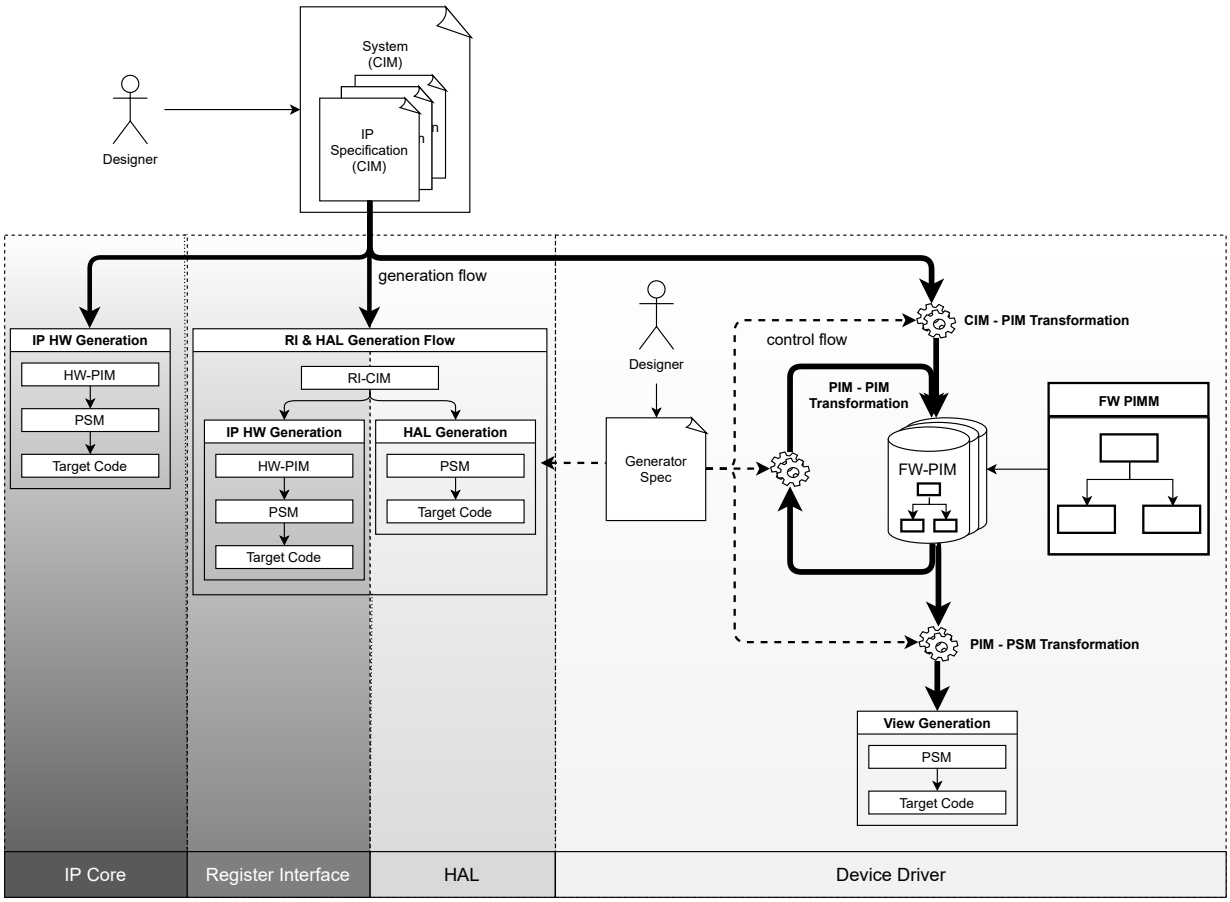


Figure 6.1: *Integration of the embedded software generation flow in the embedded design flow.*

The embedded software flow follows the same three-layered MDA approach as the HW generation flow of MetaRTL, outlined in Section 3.3.1. Both flows use device-specific generator frontends to transform the CIMs into PIMs of their domain. In HW, the generator maps the specification to a HW PIM that describes an abstract RTL architecture with various design primitives as building blocks. The high configurability of the CIM combined with a flexible HW generator enables numerous hardware variants, which exponentially grow with the number of attributes in the CIM.

Following the same concept, the parallel firmware flow provides an abstract formal description that combines the structural and behavioral view of the device driver defined by the FW metamodel. It captures both aspects for specifying the device's behavior and an object-type system tailored to firmware development. An instance of this metamodel (FW PIM) is derived from the CIM with device-specific transformations. In general, the implementation of firmware requires detailed knowledge about the underlying hardware. Due to the single-source principle of the proposed framework, the instantiated FW PIM automatically aligns to the respective generated HW variant.

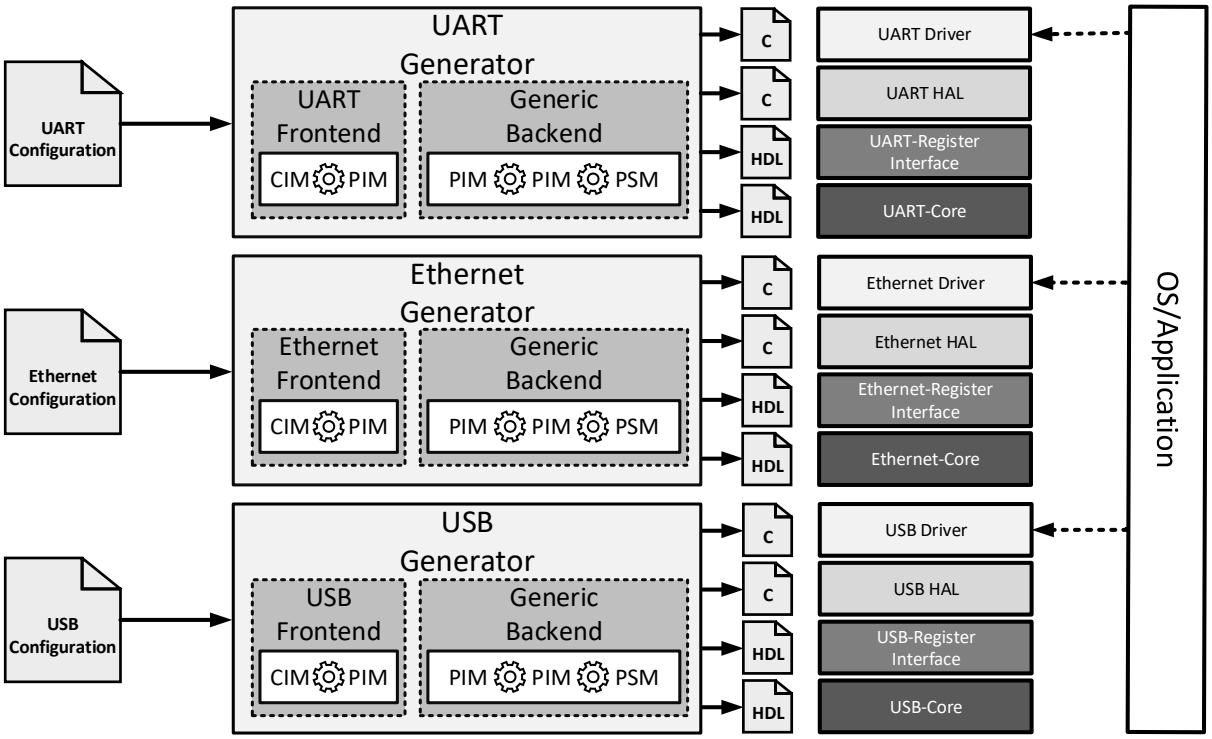


Figure 6.2: *SoC stack for different IP components.*

The device-specific transformation belongs to the framework's generator frontend, which the designer implements for each IP component individually. Instead, the generator backend is applied to each instantiated PIM in the same way, regardless of the IP. Compared to the hardware flow, the embedded software flow structures the first transformation in two steps: The first step refactors and extends the PIM (*PIM-PIM transformation*) without changing the abstraction layer or manipulating the actual firmware behavior. For example, it carries out the integration of safety measures, performs coding guideline checks (e.g., suggested by MISRA in table 4.2) and implements different driver design styles. In the second step, the PIM feeds different view generators (*PIM-PSM-Transformation*) proposed in Section 4 to realize different platform-specific target

languages and custom styles.

These generator steps are the most significant advantage of such a framework since they can be exploited to produce different manifestations of the firmware design for a particular IP configuration. For this purpose, the generator steps provide configurable transformations (mainly for the generator backend) to realize different design decisions controlled by a *generator specification (generator-spec)*. The customizable generator steps are ideal for building recurring memory-mapped IPs in SoCs, such as SPI, UART, I2C, Timer, and Interrupt Controller. The designer only needs to implement the generator frontend for each IP while reusing the generic generator backend written in a one-time effort, as shown in Figure 6.2. The design effort for a device driver[28] is thus limited, but the number of producible variants is enormous.

6.2 Abstract Embedded Software Model

The heart of the framework for firmware generation is the platform-independent metamodel for firmware (FW PIMM), which is shown in Figure 6.3. An instance of the FW PIMM is constructed, transformed, and forwarded to different view generators utilizing multiple generator steps. The FW PIMM combines aspects to capture device attributes, software architecture and behavior fully. So, it serves as an intermediate model of the embedded design flow that covers all relevant features required to generate complete device drivers.

An instance of the metamodel, the FW PIM, resembles the complete structure of any firmware target file. On the one hand, it captures the basic structural features (skeleton) of the device driver consisting of variable and function declarations. According to the software layers, this structural specification defines the application programming interface of the device driver to the upper application or operating system layer. On the other hand, the FW PIM specifies behavioral features inspired by UML activity diagrams to model processes' control and data flow (*Activities*). A designer defines an *Activity* as a programming sequence constructed by a procedural order of *Action* nodes.[29] In general, each action defines a subordinate behavior that cannot be further decomposed within the activity. The FW PIMM provides actions either as control nodes (e.g., decisions, loops) or data nodes (e.g., object manipulations, arithmetic operations) that can be arranged in any order. Conclusively, the FW PIMM covers two essential aspects to specify a complete structural and behavioral abstract firmware model: A comprehensive type system (*Objects*) and a large set of firmware-specific *Actions*. Both, *Actions* and *Objects*, are designed as templates that can be easily extended by other entities in new versions.

All firmware's target files are instantiated in the same single FW PIM in the embedded software flow. An instance comprises each IP as a *Driver*, including one or more *Modules*, also called channels. A module, in turn, specifies the structural and behavioral features of the device. This general consideration simplifies the cross-referencing of variables and functions between different scopes. Additionally, it allows performing further rule checks, e.g., concerning the visibility of objects in the firmware as defined in D4.5, D8.3.

[28] The framework is not limited to device drivers. It can also create firmware libraries that do not have an underlying hardware generator. In the following, the term device driver is used for all these kinds of the firmware.

[29] Compared to the abstract HW model, the FW model maps a sequential programming structure, similar to programming languages such as C.

Furthermore, a firmware component can optionally include a *Statemachine*. The state machine structure is similar to UML's state machine notation, consisting of *Transitions* and *States*.

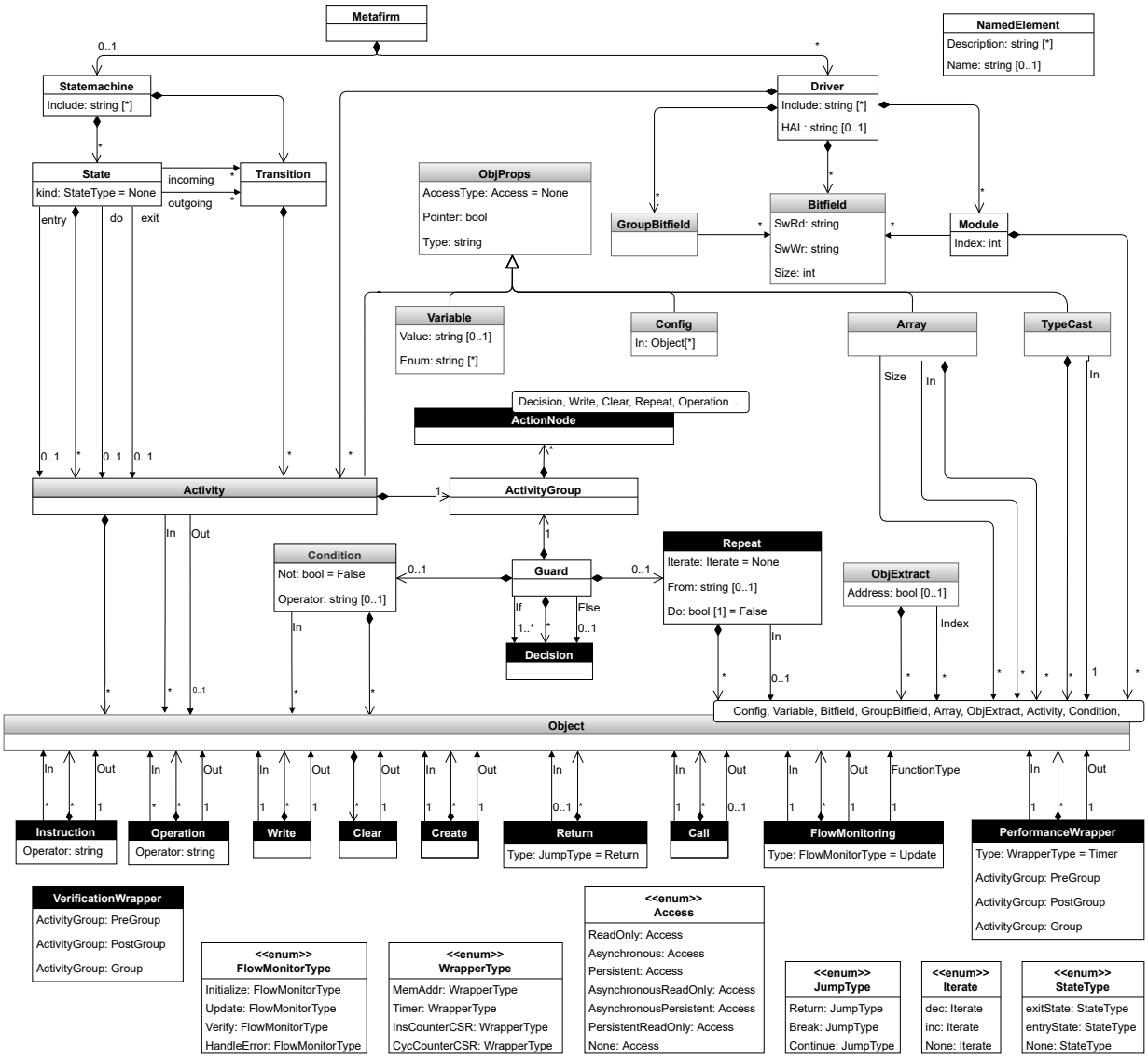


Figure 6.3: *The FW PIMM (Metafirm) of the embedded software generation framework. The gray classes depict the abstract object type system. The black boxes represent the action system that can be assembled into sequences to describe the behavior of a device.*[30]

6.2.1 Limitation

The FW PIMM is an abstract representation of conventional embedded software languages such as Misra-C or Misra-C++. Thus, it covers a wide range of embedded language constructs on an abstract level. The model is mainly considered for developing simple embedded software architectures following simple control loops, cooperative multitasking or preemptive multitasking. For two reasons, the FW PIMM is not recommended for embedded software that deals with sophisticated kernels beyond simple microkernels. First, it supports no simultaneous behavior (parallelization),

[30] Note: All metamodel classes have a generalization relation to NamedElement, which contains the attributes Name and Description. For simplicity, the generalization relationship has been neglected from the figure.

which is mainly required for modeling multicore systems with complex synchronization efforts. Second, the FW PIMM supports inline assembly capabilities only in a basic way. Accordingly, systems demanding substantial assembly parts for complex resource management or speed-critical behavior require further extensions or a special assembler submodule.

Compared to the UML activity diagram, the FW PIMM focuses on implementing synchronous processes that define a process through subsequent actions. However, the UML notation provides specific asynchronous events or invocation actions such as *Send Signal Action*, *Timer Action* or *Call Behavior Action*. The FW PIMM, instead, implements asynchronous behavior as interrupts. Furthermore, the FW PIMM defines some actions provided in UML as a conjunction of different action nodes. For example, a *Wait Action* is implemented as a decision node in conjunction with a timer call. A *Send Signal Action* is implemented as an IP-Transmitter call. However, the required UML-functions are not needed for device driver generation and are therefore not supported.

6.2.2 Objects

The object-type system (classes in gray) constitutes the central element of the FW PIMM. *Objects* define entities that can be referenced in various aspects of a firmware file. They serve as, e.g., arguments for functions or as input or outputs for various action nodes. Compared to EBNF or C-PSM, an *Object* constitutes identifiers and also expressions. In a real application, an object can be of any complexity. The FW PIMM provides a flexible scheme to create and reference all possible variants of objects in the same way. In other words, it specifies no predefined type system (e.g., int, char). However, it gives the designer the latitude to customize objects and types.

Table 6.1: *A list of different access types and the corresponding type identifiers (example for the language C) specifying objects.*

Access type	Qualifiers	Remarks
None		Object is stored in RAM and the cached value can be used.
ReadOnly	const	The variable is protected throughout the entire scope. Cached values can be used.
Asynchronous	volatile	The compiler applies no optimization and access reordering. Cached values will not be used.
Persistent	static	The variable remains in memory and preserves the value. It limit the variable's or function's scope.
AsynchronousPersistent	static volatile	Forces different threads to read the variable each time.
AsynchronousReadOnly	const volatile	The code keeps the value, but it can be changed externally.
PersistentReadOnly	static const	A protected variable with internal linkage.

Object is a generic class in the metamodel that can take different variants and is easy to maintain due to consistency. Each manifestation of the object is declared, assigned and transformed similarly. The most basic object is the *Variable* that describes a storage element of a specific type. The type and scope of each variable and certain other objects are managed by the class called *ObjProps*. This class provides a very abstract pattern to cover all needed data representations. So, the FW PIMM does not specify concrete types. Instead, it defines the *Type* as a string attribute for giving a hint that representations are the same. A variable can refer to any *ObjProps* instance, with the object defined as both default type (e.g. int, boolean) or as a complex type structure. In addition, the *ObjProps* can be used to declare the variable as *Pointer*. Furthermore the *ObjProps*

contains one entry to declare the variable as *Pointer*. Another entry is used to specify the storage and *AccessType* which can be obtained from the Table 6.1.

The objects demonstrate the essence of FW PIMM's concept of reusing model patterns as much as possible. Many other object types reuse the same attributes of (*ObjProps*) as the variable, which keeps the design effort and the static checks for the transformation low. Furthermore, a variable is defined to be assigned and reused in more complex storage elements like arrays or structures. In the following, the other fundamental manifestations of the object besides the variable are discussed in more detail:

- Bitfield: A device driver interacts with bit fields as objects. The FW PIMM describes a *Bitfield* object only with device driver-relevant attributes, including its name, software read-write capabilities and size. When instantiating the FW-PIM, a framework automatically loads all bit fields from the device-associated RI-CIM (cf. Figure 5.1), ignoring driver-irrelevant information[31]. A bit field, which is used as an entity of actions, automatically employs the functions and macros defined in the HAL. So, each *Bitfield* object's address, offset, and read and write functions can be cross-referenced.

  Each *Driver* contains a list of *Bitfields*, while a single module allocates only a subset of those. This limitation increases the robustness since each module can only address its designated bit fields. The absence of a bit field for a certain device configuration can either be exploited in the generator for describing alternative control flows or cause an error.

- GroupBitfield: The FW PIM automatically groups bit fields with the same context but different modules in *GroupBitfield*. Each group represents a pointer array for a common bit field that exists among IP modules. Groups reduce the coding effort for transformations and simplify dealing with different generator and device configurations. They comprise important information for the generation steps, which can be exploited in the generator to examine the presence and number of bit fields of a particular context.

Listing 6.1: *Three different device driver implementations for activating timer channels. Note: The code snippets show the generated driver code for a HAL that follows the implementation variant of Specific Inline Accesses of Section 5.2.2.1.2.*

```
// a): Enabling a specific timer channel via specific driver function.
void enable_timer_channel0(){
        BF_ACTVAL_CH0_WRITE(MAX_VALUE[0]); // HAL: Reset actual counter value.
        BF_STATUS_CH0_WRITE(TIMER_READY_ENUM); // HAL: Status of the timer channel.
        BF_ENABLE_CH0_WRITE(1); // HAL: Enables the timer channel.
}

// b): Enabling a specific timer channel by function argument via generic driver function.
void enable_timer_channel(TIMER_CHANNEL_ENUM CH){
        Timer_HAL[CH].BF_ACTVAL_WRITE(MAX_VALUE[CH]); // HAL: Reset actual counter value.
        Timer_HAL[CH].BF_STATUS_WRITE(TIMER_READY_ENUM); // HAL: Status of the timer channel.
        Timer_HAL[CH].BF_ENABLE_WRITE(1); // HAL: Enables the timer channel.
}
```

[31]The designer of a transformation script can also directly reference the complete RI CIM with all bit field attributes. In this case, however, the designer has to utilize the API of the register interface metamodel to access its attributes.

```
// c): Enabling all timer channels via loop.
void enable_timer_channels(){
    for (int i=0; i<TIMER_CHANNEL_MAX; <++){
        Timer_HAL[i].BF_ACTVAL_WRITE(MAX_VALUE[i]); // HAL: Reset actual counter value.
        Timer_HAL[i].BF_STATUS_WRITE(TIMER_READY_ENUM); // HAL: Status of the timer channel.
        Timer_HAL[i].BF_ENABLE_WRITE(1); // HAL: Enables the timer channel.
    }
}
```

A designer describes the behavior in the transformation by referring to the bit field type (e.g., CHANNEL_EN) rather than a specific bit field (e.g., CHANNEL_0_EN). This simplifies the design of generator patterns and enables the handling of different device driver designs in a single transformation, as shown in Listing 6.1. In *a)*, the specific device driver implementation addresses particular bit fields of the groups directly. In contrast, the implementation *b)* generically (e.g., via a subscript) writes the bit fields of the group. Furthermore, Listing 6.1 shows addressing the bit fields of all groups via loops as provided in the code snippet *c)*.

- Config: A *Config* is comparable with a structure in C. It contains a list of various *Objects* that recursively may contain further *Configs*. Thus complex hierarchical structures can be defined, necessary, e.g., for the specification of device handlers.

- Array: The *Array* describes a multidimensional list of objects. All entries of the array have the same *ObjProps* type as the array type itself. In general, the structure of the array resembles that of the *Config*. Nevertheless, an array additionally references a list of objects to describe its dimensions. Accordingly, an object as an array entity is either intended to define the array *Size* or an entity of the array (*In*). The objects that specify the array's dimension sizes can be of any type, e.g., a bit field. All other objects are subject to static type checks that verify the array criteria of matching component types.

- TypeCast: The *TypeCast* is a special object mostly considered an expression in a target language. However, it is treated as an object in the FW PIMM since it has no impact as a standalone action. It converts an object of one data type into another. A *TypeCast* takes a single object as input and defines an explicit conversion to the new type given by *ObjProps*.

- ObjExtract: Similar to the *TypeCast*, the *ObjExtract* takes one object as input. This object is applied for two different purposes. First, for indexing specific elements of an array. The indices reference a list of objects to access the array element (in FW PIMM: *Index*).

  Second, the *ObjExtract* is used to reference or de-reference objects. Referencing, i.e., setting the attribute *Address* to True, means taking the address of an object. De-referencing (*Address* is False), on the other hand, takes the value pointed to. Exemplary for C code, the reference operator (&) or de-referencing operator (∗) is thus inserted into the generated code.

- Condition: A *Condition* is primarily a part of a conditional statement, e.g., if-statement. The FW PIMM handles the *Condition* as a non-standalone entity that can only be referenced within actions. An object defined as a condition is always evaluated as either true or

false. A designer can define a unary condition by referencing a single object, as shown for `n_full_cond` in Listing 6.2. Alternatively, a condition may be described by two objects linked through an *Operator*, defined in `count_cond`. For more complex conditions involving more than two objects, e.g., `spi_write_cond`, the designer must specify condition trees using recursive references to other conditions.

Listing 6.2: *Specification of a nested condition from the SPI driver generator.*

```
// Expected/Generated Code: ((! bitfield_get(BF_FIFO_FULL)) & (i < max_count))
count_cond = sCondition(Ins=[i, max_count], Operator="<"); // ( i < max_count)
n_full_cond = sCondition(Ins=[BF.FIFO_FULL], Not=True); // (! bitfield_get(BF_FIFO_FULL);
spi_write_cond = sCondition(Ins=[n_full_cond, count_cond], Operator="&");
```

- Activity: An *Activity* as an object specifies a function defined within the scope of the FW PIM. In case the activity is defined as a *Pointer*, the object provides the pointer to the function. Otherwise, the object is interpreted as a function call.

All introduced objects are processed equally within the framework. For example, actions always define operations taking predefined objects as entities independent of their type. The decomposition of the objects into target code is hidden from the designer and takes place in the generator back-end. Object construction is accomplished utilizing the auto-generated API as shown in Listing 6.3. On top of the API, object handling (e.g., instantiating and static checks) is extended and optimized by the FW DSL detailed in Section 7.2.

Listing 6.3: *A set of examples demonstrating the initialization of objects utilizing the automatically generated API. Note: The FW-DSL provides simplified functions serving the same purpose.*

```
/*Example Variable: uint16_t *rx_data */
rx_data = sVariable(Name="rx_data", Pointer=True, Type="uint16_t");
/* Example Array: uint16_t rx_data_array[bitfield_get(BF_FIFO_SIZE)] */
data_array = sArray(Name="rx_data_array", Sizes=[BF.FIFO_SIZE], Type=rx_data.getType());
/* Example ObjExtract: rx_data_array[2] */
data_array_idx2 = sObjExtract(In=data_array, Index=[2]);
/* Example TypeCast: (uint8_t) tx_data_array[2] */
data_reduced = sTypeCast(Type="uint8_t", In=data_array_idx2)
/* Example Activity: ctrl_uart(); */
ctrl_uart = sActivity(Name="ctrl_uart", Type="void")
/* Example Config: UART_Config_t UART_Config = {data_reduced, *rx_data} */
config_data = sConfig(Name="UART_Config", Type="UART_Config_t");
config_data.addObject(data_reduced);
config_data.addObject(rx_data);
```

6.2.3 Activities and Actions

The second part of the FW PIMM deals with functions and their behavior. A target file represented as a *Driver* entity provides a list of n-ary functions (in FW PIMM: *Activity*), taking n objects as arguments. Each function is perceived as a self-contained activity within the scope specified by the *AccessType*. Note: One activity specifies the function header, while all activities of a driver entity provide the complete driver API.

The structure of the behavior follows the structured programming paradigm since it maps the abstract behavior in the same order in which it is created and also executed. Considering the FW PIMM, the behavior or control flow of an activity is given by the *ActivityGroup*, which symbolizes a cohesive basic building block with a sequential list of action nodes. An *ActionNode* also called an action, is a single step within the control block that is not further decomposable at the PIM layer. The FW PIMM defines actions as a template class that can take various types, similar to objects. In order to realize a variety of different behaviors, these action nodes can be freely arranged to a coordinated flow of actions. In other words, the FW PIMM maps a control flow as a flexible arrangeable list of actions. This list notation has two significant advantages: First, Python's built-in functions for list population or list manipulation can create or transform the control flows. Second, a list reflects an exact sequence of behavior with a start and end node and enables the control flow analysis at an abstract level.

The structure of all action entities (classes in black) has some common characteristics since they always reference a set of objects to realize a certain behavior.[32] Considering the FW PIMM, an action references an object as an incoming (*In-*) or outgoing (*Out-*) object. As a straightforward analogy: An In-object serves as the input of action and typically retains its value, e.g., operand on the right-hand side of an assignment. Out-objects, in contrast, are modified by the action (left-hand side).

In general, the actions are clustered into two different groups. One group comprises control flow actions that coordinate the flows through alternative sub-control blocks. The other defines object actions that purely map the data flow.

6.2.3.1 Object Actions

A list of object actions is treated as an ordered sequence of successive behavior steps without branching (An exception is the *Return* action). The sequence must be strictly maintained through the different transformation steps. In other words, manipulating the list by injecting action patterns is legal unless it changes the order of the existing actions.[33]

An object action in the FW PIMM describes the smallest single step, which can not be further subdivided. Such a step describes an action on an object, e.g., *Write*, *Clear*, *Create*, *Operate*. The notation of all actions is uniform, thus reducing the complexity of the abstract language and leading to more usability and less implementation effort for transformations. All these actions share an optional composition to objects, which is the key aspect of object-based actions. This freedom allows referring to an object that is not in the scope of the FW PIM or is not instantiated, e.g., a constant or an external object. In addition, each action refers to one or more objects, depending on the action's type. At this point, the action can associate its composite object or any other object in the scope.

As shown in the examples, the structure of all object actions is similar, thus increasing usability and reducing the implementation effort of generators.

[32] This consistent characteristic of the template-based design of actions in FW PIMM is suitable for extensions so that new actions, e.g., safety-related actions, can be easily added to the FW PIMM.

[33] However, the binary code or the final execution sequence may differ. A compiler can rearrange the order, guaranteeing that the optimization does not change the functionality.

Listing 6.4: *A set of examples demonstrating the implementation of object actions utilizing the generated API. Note: The FW-DSL provides simplified functions serving the same purpose.*

```
1  //Variable declarations from Listing 6.3
2  /*Example: Create */
3  group.addCreate(Out=rx_data);        // uint16_t *rx_data;
4  group.addCreate(Out=rx_data_array);  // uint16_t rx_data_aray[bitfield_get(BF_FIFO_SIZE)];
5  /*Example: Write */
6  group.addWrite(Out=rx_data, In=rx_data_array);  // *rx_data = rx_data_array;
7  group.addWrite(Out=rx_data, In=BF.RX_DATA);     // *rx_data = bitfield_get(BF.RX_DATA);
8  /*Example: Clear */
9  group.addClear(Out=rx_data);    // *rx_data = 0;
10 /*Example: Return */
11 group.addReturn(Out=rx_data);   // return *rx_data;
12 /*Example: Operation */
13 group.addOperation(Out=rx_data, Operator="++");   // *rx_data++;
14 /*Example: Compound Operation:   data = data << (32-data_width);*/
15 group.addOperation(In=[data, dummy], Out=data, Operation="<<");    // data = data << dummy;
16 group.addOperation(In=[32, data_width], Out=dummy, Operation="-"); // dummy = 32 - data_width;
17
18 /*Example: Instruction  */
19 //  __asm__ __volatile__( "mv %0, %1" : "=r"(rx_data_aray[2]) : "r"(*rx_data));
20 group.addInstruction(In=[rx_data], Out=[rx_data_array[2]], Operator = "mv");
```

The implementation of several actions is demonstrated in Listing 6.4. All examples reuse objects constructed as shown in previous Listing 6.3. An essential object-action class is the *Create* action, which declares an Out-object in the sequence. In lines 3 and 4, the Create action initializes a pointer variable and an array. However, it can be applied to any other object type. Optionally, the node can be extended by an In-object that pre-initializes the Out-object.

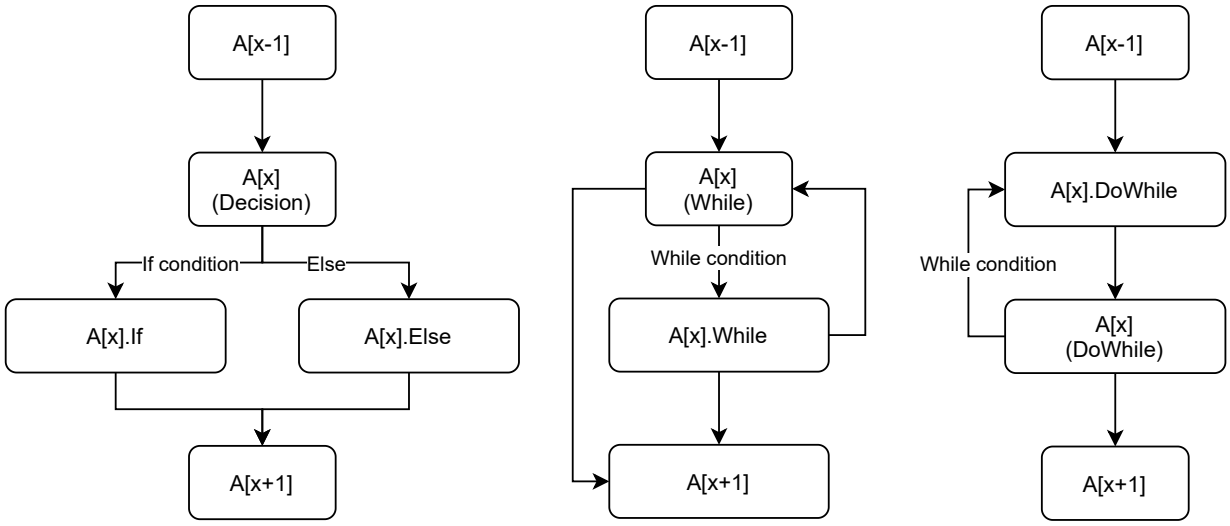
However, the Write action overwrites an existing Out-object with an In-object. As stated in lines 6 and 7, it is mandatory to specify exactly one incoming and outgoing object. Keep in mind that the generated code for a *Write* actions automatically adapts to each object combinations, e.g., $Object_{Bitfield} = Object_{Activity}$, $Object_{Variable} = Object_{Config}$. In contrast to the *Write*, the *Clear* action refers only to the Out-object being deleted or destroyed. Similarly, the *Return* action serves as an output node for the activity or action sequence that returns an optional Out-object.

Operations are arithmetic or logical calculations with one operator and an arbitrary number of input objects. So they can appear as an unary operation without an incoming object (line 13) or as an n-ary operation with n In-objects. In this context, complex operations with multiple operators can be constructed as compound operations complying with the expression system of the VLD (see 4.4.3). An example of a compound operation is shown in lines 15 and 16. Note that the DSL simplifies the construction of compound operations in a single function.

Existing UML specifications for embedded development have ignored the unification of low-level and high-level code in abstract models. The FW PIMM solves this shortcoming and supports a low-level *Instruction* action that follows the same structure as the *Operation*. The *Instruction* allows embedding special low-level code segments into the control flow, which is important for two reasons: First, it can result in more efficient or safer code than the compiler might otherwise create. Second, particularly in the domain of RISC-V systems, processor-specific instructions can be incorporated into the control flow.

6.2.3.2 Control Flow Actions

The activity's control flow actions are executed in the order in which they appear in the action list. Control flow actions or control nodes, however, break this linear control flow into alternative execution orders. The FW PIMM considers an action as a control node if it inherits one or more sub-control flows as basic blocks. Often, this is also associated with branching and enables the design of conditional executable basic blocks. The three most frequently used control nodes (while, do-while, if-else) are shown in the control flow graph (CFG) of Figure 6.4.



(a) *if-else* (b) *while* (c) *do-while*

Figure 6.4: *The control flow graphs for three standard control nodes available in the FW PIMM.*

A *Decision* or if-then control node specifies a nested control block executed once under a certain condition. In the FW PIMM, an if-decision comprises a Guard (If), which contains a *Condition*-object and the conditionally executed action list (*ActivityGroup*). Furthermore, a decision node can be extended by a set $n_g \in \mathbb{N}$ of various guards, each providing a different execution path. In this case, the if-guards $g_{if} = \{1...n_g - 1\}$ must contain a condition. At the same time, the last guard $g_n = \{n_g\}$ can optionally be specified as an alternative *Else*-guard without condition. The CFG in *a)* illustrates such an if-then-else scenario. In this example, an action list A contains a control node at position x that branches into two distinct execution paths. Each comprises an action list $A[x].If$ and $A[x].Else$ as a child.

Compared to the *Decision*, a *Repeat* or loop node continually executes one building block recursively. In FW PIMM, the *Repeat* is specified by a single *Guard* and thus by a unique *ActivityGroup* and *Condition*. The class can be instantiated in different ways to create several types of loops. However, all loops have one thing in common: once the condition is no longer met, the loop terminates and continues executing the parent action list.

The CFGs in *b)* and *c)* show two characteristic loop structures, a while loop (same control flow as a for-loop) and a do-while loop. Both loops have identical structures but have either the attribute *Do* enabled or disabled. Following the graph, a do-while loop at position x of an action list A comprises a block $A[x].DoWhile$, which must be executed at least once before executing $A[x+1]$. A while-loop instead can directly jump to the action $A[x+1]$. In contrast, the for-loop

requires additional attributes. Besides the end condition, a for-loop declares an *In*-object as a loop control variable and an *Iterate* that defines the increment of the control object.

Listing 6.5: *Initialization of a control flow with multiple branches.*

```
 // if (SPIConfig.Receiver){
if_group = group.addIf(Condition=self.Receiver) // Child list "if_group"
if_group.addCreate(...) //if_group[0]

// while(!bitfield_get(BF_FIFO_EMPTY)){
while_group = if_group.addRepeat(Condition=[BF.FIFO_EMPTY], Operator="!") // Child list "while_group"
while_group.addWrite(...) //while_group[0]

// if(!(bitfield_get(BF_FRAME_ERROR) | bitfield_get(BF_PARITY_ERROR))){
frame_parity_error = sCondition(Ins=[BF.FRAME_ERROR, BF.PARITY_ERROR], Operator = "|")
not_error_group = while_group.addIf(Condition=[frame_parity_error], Operator = "!")
act_search = not_error_group.addWrite(...) //not_error_group[0]
// }}}
```

A complex device driver function with many alternatives to support may contain multiple nesting levels, each providing a new action list as a leaf and thus resulting in an action list tree. Regardless of the nesting level, the designer uses the same API to assemble, analyze, and transform the particular action lists of the tree. An abstract implementation example for such a code with multiple nested branches is shown in Listing 6.5. An action list or individual actions within this tree can be accessed via index operators since the framework supports easy handling of multi-nested control blocks by exploiting Python's NumPy array indexing. Thus, the action named `act_search` from the example can be accessed from different tree perspectives, e.g. `not_error_group[0]`, `while_group[0][0]`, `if_group[0][0][0]` or `group[0][0][0]`. This makes the implementation of the generators more straightforward.

The FW PIMM provides two other control blocks named *VerificationWrapper* and *PerformanceWrapper*, presented in detail in sections: 9.3.2 Performance Analysis and 9.3.1 Firmware Verification.

6.3 Generator Specification Model

A generator frontend is implemented to not include any specific implementation feature into the FW PIM. Accordingly, the transformations focus purely on the functional aspects of the firmware. Nevertheless, there is a strong demand to support different system and application requirements by different design choices. For this purpose, the generator specification metamodel is introduced in Figure 6.5.

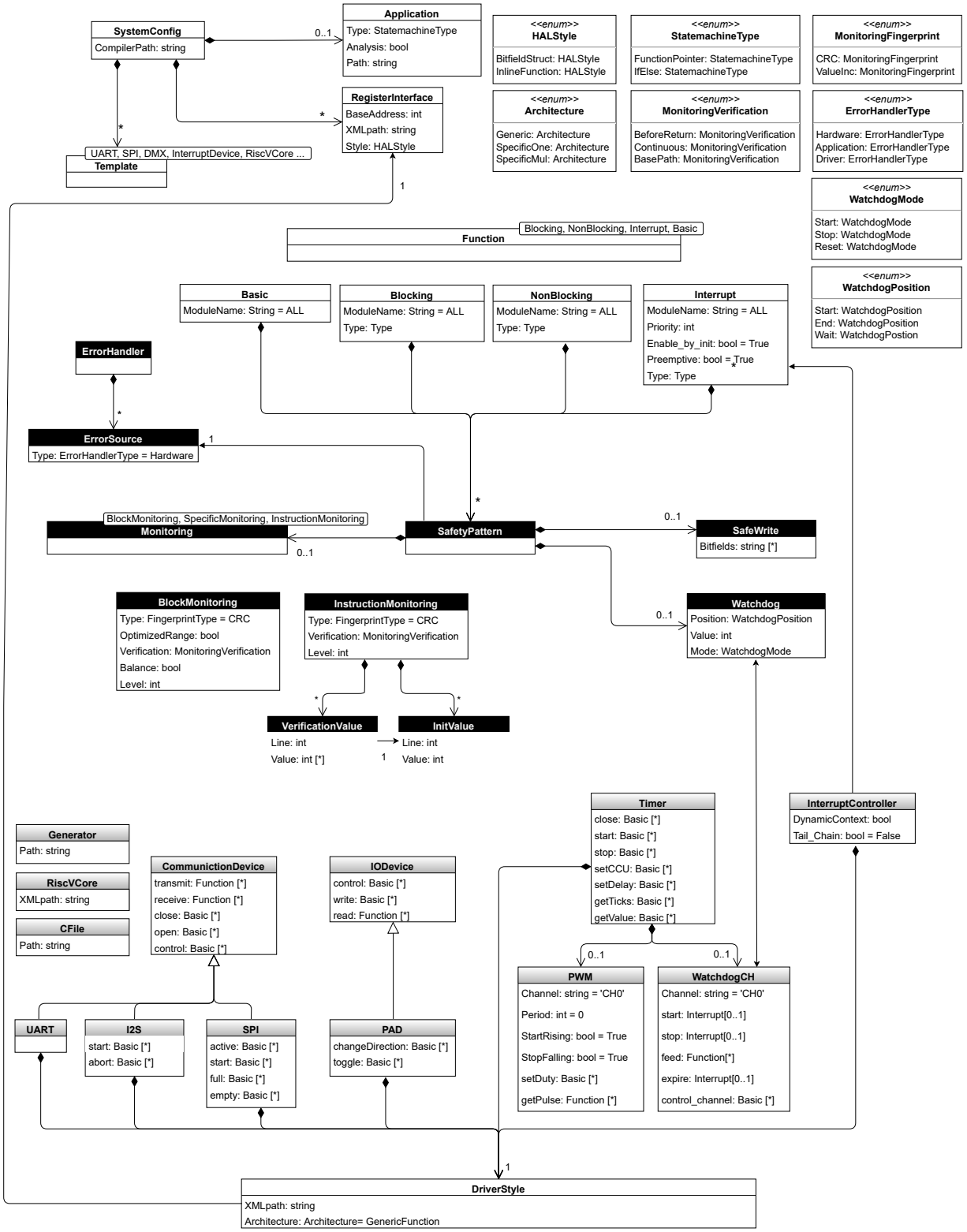


Figure 6.5: *The generator specification metamodel (generator-spec) controls the generator steps. The metamodel organizes the software architecture (gray classes) and includes optional safety features (black classes). The figure shows the reduced metamodel, which contains a subset of all available templates.*

This metamodel serves as a user interface whenever a designer customizes a new software architecture. A single instance of this metamodel is called a generator-spec, which drives the entire generation process with all generators involved. It primarily specifies the software architecture with all its components. Respectively, it stipulates those IPs that have to be built, respectively, those IP generators that have to be executed. Furthermore, the generator-spec defines the design options and the API of these components. The design options include features to customize the implementation style and optional non-functional transformations (e.g., safety patterns) for each component.

The next two chapters discuss the transformation implementations of all these settings, while the generator specification metamodel is discussed in more detail below.

6.3.1 Organization of the Software Architecture

An SoC assembles multiple IPs provided by the framework. The generator-spec organizes all IPs and firmware components used in the embedded system through the *SystemConfig* class. In general, it manages the *Paths* to all CIMs (given as *XMLPath*) and transformation scripts of the project. Accordingly, it executes all required generators to implement the desired embedded system, consisting of the following sub-components.[34]

- The *Application* describes the main routine of the embedded system. It is referencing the transformation script that assembles the FW PIM. Especially in polling-based designs, the application defined as a state machine main routine. The generator-spec provides two *Types* of state machines. First, a function-pointer-based state machine containing a state transition matrix or an if-else-based state machine.

  A designer must be aware that the preferred solution largely depends on the application and system requirements. An if-else-based state machine has more overhead with the conditional statements for an increasing number of events and states, as stated in [8]. Accordingly, it is not intended for time-critical applications. The table-based design, however, leads to more memory overhead.

- The *RegisterInterface* controls the HAL generator. It defines the implementation style of the generated HAL as either (Specific Inline Accesses) or (Generic Bitfield Structures). In addition, the generator manages the address ranges through *BaseAddress* for each interface to avoid overlapping address spaces.

- A *Template* can define either a component specified through an IP CIM or a fixed firmware file (e.g., a library file) that is not configurable by an abstract specification. In the Figure 6.5 of the metamodel, these classes are highlighted in grey. An IP CIM can describe a hardware module of the SoC which is subject to an HW generator that results in a device driver when executing the generation flow.

  Each component, e.g., *I2S*, *SPI*, *PAD*, is adaptable to a set of requirements. For example, a designer can configure not only the *Style* of the driver but also each function individually.

[34] The metamodel is designed so that further generators for IPs or software components can be easily added.

For instance, the *I2S* features a set of functions such as *receive*, *transmit*, *open*, *start* and others, which altogether constitute the driver interface of the I2S.

Likewise, a template can also be a CIM that only feeds the firmware generator and not a hardware generator. In other words, it is a pure software component, such as a ring buffer or a data scrambler. Additionally, the generator-spec is flexible as it provides further generic templates to enable a wide range of applications. First, it can include *Generators* that have no CIM as input resulting in a non-configurable and static PIM instantiation. Second, a designer can incorporate *CFiles* as external libraries which can be referenced by the generator, see 9.3.

6.3.2 Generator Configuration Settings

A main advantage of the framework is its ability to generate different design and architecture variants for the same embedded system and software architecture. The generator-spec model provides several options to create a wide variety of implementations. Note that each variant retains the underlying behavior. This design flexibility offers two advantages. On the one hand, the generator and the resulting firmware design can be adapted to different application and system requirements. On the other hand, the design diversity enables analytical methods to examine variants by SW cost, e.g., memory footprint.

6.3.2.1 Device Driver Architecture and Design

Firmware can be designed and implemented in many different ways. Every design decision can impact not only the efficiency of the firmware but also the real-time behavior and memory footprint of the system. The driver *Architecture* is one design principle that must be chosen individually for each device driver. Architecturally, a generic and specific implementation style is distinguished.

A generic architecture provides a driver implementation that supports all instantiated modules of the IP MoT. Such a driver is designed in a minimalistic way to take every potential behavior into account[35]. In other words, the driver can be reused for every module of the peripheral type in the system. However, a generic device driver may not be as efficient as a specific device driver. More control blocks are generated with increasing diversity between modules, slowing down the generic design's performance. Indeed, a specific driver adapts to one specific module variant and therefore requires less time-consuming conditional control flows in the implementation. Nevertheless, as a consequence, more instruction memory is occupied (.text section) since each module requires a specific implementation. The generator-spec provides two options: *SpecificMul* creates a separate driver target file for each module, while *SpecificOne* packs all module-specific entries into one file.

A second design decision concerns the architecture of the individual driver functions. Certain functions that mainly interact with external components, e.g., I/O functions, can be implemented in the following modes, as shown in the sequence diagrams of Figure 6.6:

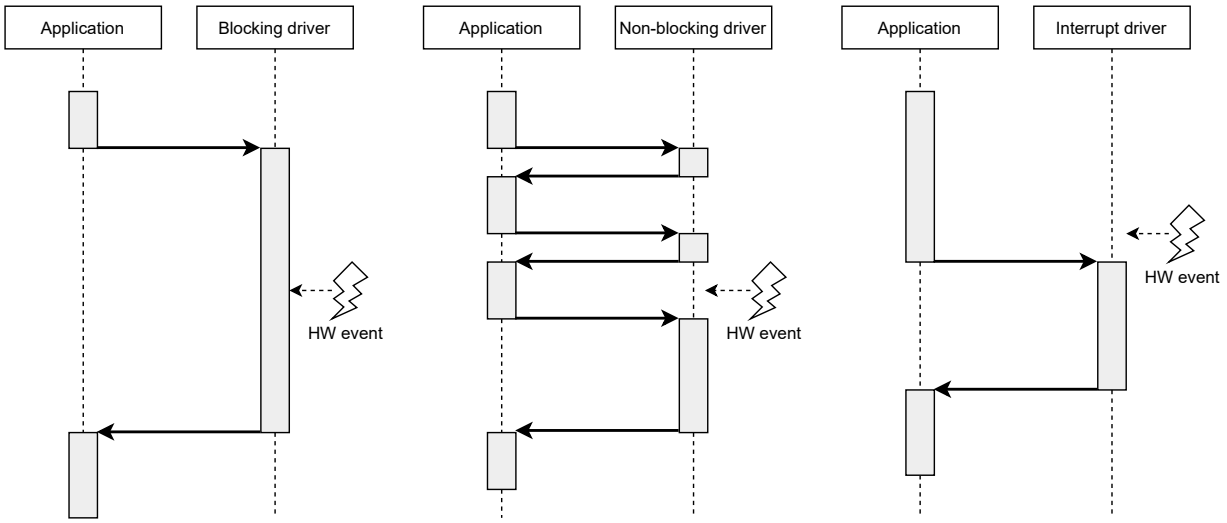
- A *blocking* architecture blocks the CPU until the process completes. A blocking function may enter a busy-wait, waiting for a specific hardware event to continue the operation.

[35] A generic device driver created for a specific set of modules of a certain IP MoT is valid for this set. However, this does not imply that it can be applied to any other IP MoT configuration since the generator optimizes the generic design for the specified CIM.

During this time, the driver is continuously polling a register, which consumes power and devours cycles that could be used for other operations. For example, for a communication device like the SPI, the execution time of the blocking function is thus dependent on the baud rate of the SPI interface.

- A *non-blocking* driver function returns immediately once the hardware is unable to continue the execution. When the driver function is left, the driver status is returned to the application to be resumed later. In the meantime, other portions of the application can access the microprocessor. In fact, this increases the application's performance but causes non-deterministic behavior, which may pose a hazard.

- An *interrupt*-based driver function is called an ISR (Interrupt Service Routine). A function can only be generated as an ISR if the SoC provides an *InterruptController*. The detailed implementation of the ISR is strongly dependent on the interrupt controller's specification (e.g., non-vectorized or vectorized, nested interrupts ...) detailed in Section 8.1.2.2. Interrupt-driven drivers can be more efficient because they spend no time waiting for a device to be ready. Apart from the HW overhead, the firmware's complexity also increases since the CPU has to do additional work to process interrupts and to continue the previous execution of programs (e.g., Context Switch).



(a) *blocking* (b) *non-blocking* (c) *interrupt*

Figure 6.6: *The sequence diagrams for a blocking, non-blocking and interrupt based driver function.*

Every design decision has its legitimation. Selecting the perfect configuration for a particular application and system involves a trade-off between memory requirements and performance. Time-critical applications benefit more from a specific architecture with interrupt-based functions. In contrast, a generic design with blocking or non-blocking functions is preferable for memory-constrained software.

6.3.2.2 Device Driver API

A driver API is an application programming interface that defines the high-level interface of the IP's behavior and capabilities. In general, an API should be designed so that it is generic and implementation-independent. An application developer does not focus on the hardware details and the function behavior but on the inputs required to achieve the desired results. CMSIS [83] and AUTOSAR [23] provide two important standards that describe vendor-independent interfaces for processors and peripherals. However, these standards impose a significant overhead for many applications [33, 160] and do not offer the capability for fine-tuning.

In contrast, a generator-based design is intended to reduce the overhead while supporting different specifications. The generator-spec combines these two requirements. It provides a standard API skeleton, which can be further customized to meet specific requirements.

The API of each IP driver depends on several factors. First, the choice of the desired routines from an existing collection. Second, on their design and architecture. Third, on an input type that acts as a function argument influencing the behavior of the function. Thereby, the type setup is limited to the basic data types (*uint8_t*, *uint16_t* and *uint32_t*) and their array or pointer representation (*uint8_t **, *uint16_t ** and *uint32_t **). An array type is mostly associated with an iterative process in the driver. In contrast, a basic data type leads to a single iteration.

Depending on the configuration of these settings, the behavior is enormously changed, whereas the interface description is only slightly adjusted. For example, a UART's *transmit* function can result in different APIs, as shown in Listing 6.6.

Listing 6.6: *Initialization of a control flow with multiple branches.*

```
 // Blocking and specific architecture with uint16_t as input argument.
UART_STATUS uart1_transmit_uint16_blocking(uint16_t data);

// NonBlocking and generic architecture with uint8_t* as input argument.
UART_STATUS uart_transmit_uint8ptr_nonblocking(UART_ENUM idx, uint8_t *data);

// Interrupt with uint8_t as hidden input argument.
void __attribute__((interrupt)) uart_transmit_uint8_isr(void);
```

6.3.2.3 Safety in Embedded Software

Embedded systems operate in a variety of environments that can cause undesirable interference and malfunction. These failures can be detected or even corrected by safety measures. The framework's strategy is to provide a set of safety measures as configurable design patterns that can be incorporated into any function or activity within a FW PIM. Each safety mechanism targets a specific type of failure and can be individually configured for each software component. Since the mechanisms are implemented as model transformations on the abstract layer, they are function-agnostic and more resistant to implementation errors. Furthermore, the modular approach simplifies development because the designer of a new generator can focus solely on implementing the functional behavior.

The generator-spec defines a *SafetyPattern* as an optional extension of a function that can take various forms. *SafeWrite*, *Watchdog*, *Monitoring* are three of many safety measures that can

be integrated into the design. The details of the transformations and the attributes of the safety patterns are discussed in Section 8.2. The *SafeWrite* is a simple but efficient method that verifies the correctness of a hardware register or bit field manipulation through a subsequent cross-check.

The watchdog timer is a crucial component in safety-critical systems to protect the time-related program flow [22, 143]. A watchdog in the generator-spec is linked to a timer channel of the general-purpose timer or the RISC-V counter, which raises an error (interrupt) when it expires[36]. An activity including the watchdog triggers an event that may reset, start or stop the watchdog channel. A designer can thus schedule different application tasks.

Program Flow Monitoring (PFM), however, is devised to check the correct execution of a function. Two PFM concepts are distinguished, block *BlockMonitoring* (Block-PFM) and *InstructionMonitoring* (Instruction-PFM). Block-PFM is a software-based approach that executes checksum calculations (e.g., CRC) via software commands across different control blocks. This approach ensures the correct execution sequence of the control blocks. However, the sequence of instructions within a control block may be incorrect. An instruction-PFM can handle this fault model since it calculates the checksum on each executed instruction via a hardware PFM module. In general, the instruction-PFM requires more hardware resources but is more efficient and has a lower memory footprint. Both concepts verify the executed path by comparing the resulting checksum state against the expected one. A corrupt sequence (e.g., missing checksum) will raise an error. Depending on the configuration of the PFM, the diagnostic coverage and the cost may vary.

Each safety pattern also references an *ErrorSource* grouped in a central *ErrorHandler*. An *ErrorSource* defines the course of action for faulty behavior. Multiple safety patterns can thus refer to the same error handler source, respectively fault handling response. The details of the safety pattern concepts and their implementation are covered in Section 8.2.

6.3.2.4 Analysis and Verification

Before the release of an embedded system, it has to pass detailed analysis and verification steps. The generator-spec includes a switch called *Analysis* that activates the debugging and analysis mode. In this mode, predefined analysis and verification nodes are injected into the application to examine the embedded system's correctness and performance.

Verification nodes are used as data monitors that record the status of objects and memory locations. This status is compared with a golden data set, which may result in a deviation that is treated as erroneous behavior. Also, profilers are used as analysis nodes to measure the time spent by the CPU on individual sub-behaviors. In this way, the developer can identify bottlenecks in the application for the respective SoC.

On the one hand, invoking the debugging mode helps the designer to verify different software concepts generated by the generator-spec for a given SoC. On the other hand, it eases design exploration to identify best-performance software design variants for the system configuration.

[36] A watchdog pattern requires a SoC with an interrupt controller and a general purpose timer or RISC-V counters. Otherwise the watchdog implementation will fail.

Chapter 7

Device Specific Generator Frontend

The generator front-end serves as a translator, specified as transformation rules that turn a CIM IP, which is part of a system, into the corresponding abstract firmware model. Based on the IP's specifications, the transformation rules specify a control flow that constructs different manifestations of (e.g., functions) the FW PIM based on the IP's specifications. It makes up the generator part that is implemented for each IP.

For building the generator frontend, the designer implements a list of transformations, generating different parts of the driver file. The transformations are specified in such a way that they can cope with different hardware characteristics. In general, this means that the IP front-end designer cannot focus on a single design but must cover all possible combinations of IP specifications. This way of programming is called meta-programming, which is more demanding than manual programming of a single variant. Additionally, the coding effort increases with the number of adjustable parameters.

This framework, however, reduces the overall implementation effort. On the one hand, by adhering to the single-source principle and the idea of separation of concerns. On the other hand, it offers an innovative FW DSL [86] as well as a design pattern library, as shown in Figure 7.1.

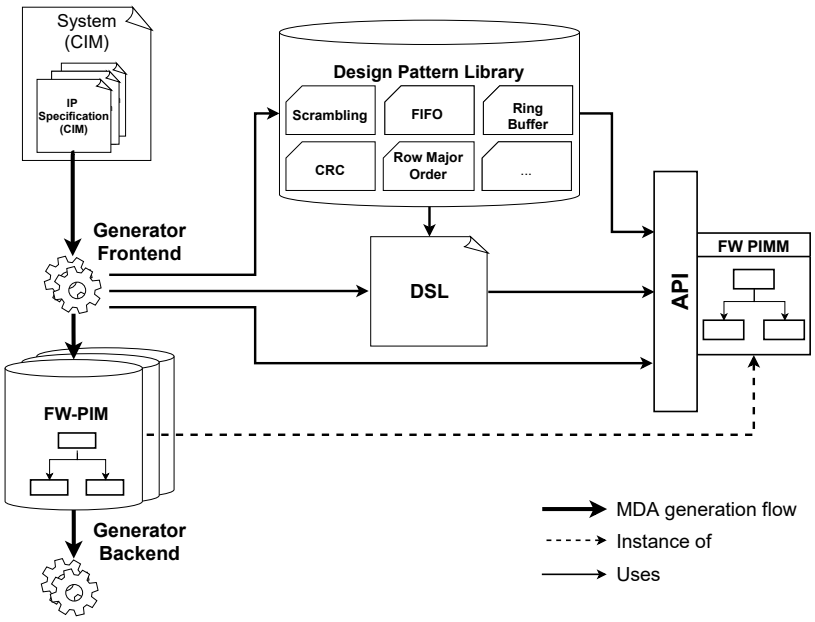


Figure 7.1: *The device specific generator frontend flow.*

The FW DSL is a partially automated extension of the automatically generated API of the FW PIMM. While the API exclusively provides standard functions for instantiating and evaluating UML class models, e.g., getter and setter, the DSL provides more advanced methods. The DSL uses the API to represent methods that describe more complex access sequences. For example, a DSL function may specify a combination of actions to realize advanced sub-behaviors, e.g., a busy-wait, read-modify-write, or conditional-return.

Instead, a design pattern describes a parameterizable transformation, utilizing the API and DSL to describe cohesive software fragments. These fragments are more complex software constructs that are typically subject to a hardware sub-component of the IP, e.g., FIFO, ring buffer or CRC. Accordingly, a design pattern must define a coherent hardware transformation (CIM IP to HW PIM) and software transformation (CIM IP to FW PIM) to include the pattern in the IP design. As the design pattern is intended to be derived from the IP CIM and reused across different IPs, two key considerations must be met. First, the attributes of the design pattern must be consistently incorporated into the different CIMs. Second, the design pattern must be incorporated into the design without further transformations or modifications.

Due to the innovative FW DSL and elegant use of design patterns, the framework significantly reduces the complexity of meta-programming. Indeed, for some IPs, the effort between generator coding and manual HDL coding remains comparable.

7.1 Template of Embedded Software

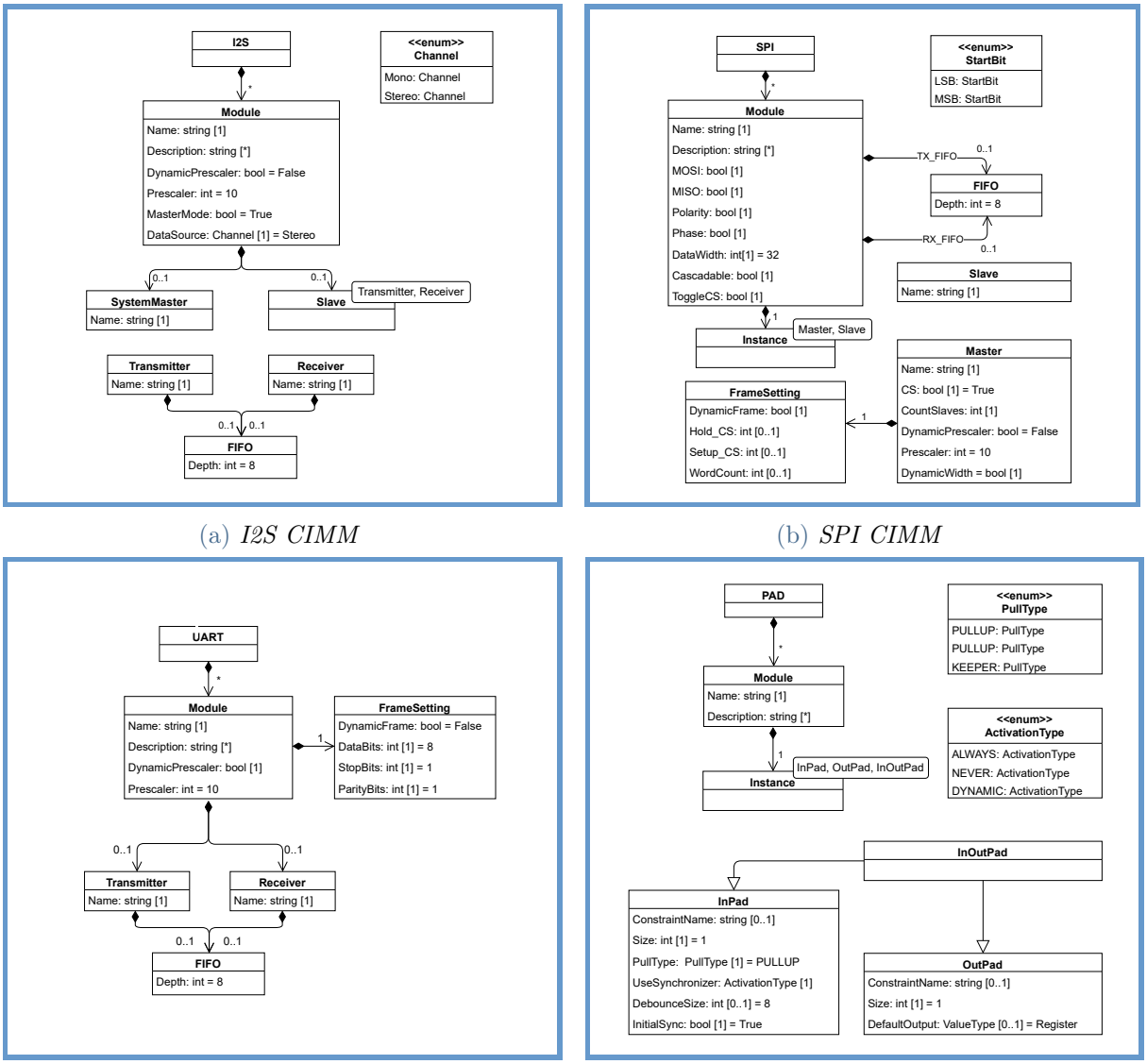
An SoC consists of various IPs, including I/Os-, system-, memory-, or communication devices such as Watchdog, Timer, I2C and ADC. Each IP is specified through its respective computational independent metamodel (CIMM), as shown in Figure 7.2. These metamodels include numerous configuration parameters that define the elementary features of the design. With an increasing number of parameters, the number of configurable variants even grows exponentially. So, the CIMM for the UART device, as shown in Figure 7.2c, contains a wide range of attributes for defining, e.g., the communication protocol, transmit/receive logic and internal storage.[37].

Writing reliable and efficient device drivers require an in-depth understanding of the underlying behavior and structure of the generated hardware. In order to cope with all variants, the framework includes the firmware generator frontend, a template designed in the same manner as the template for hardware modeling. For example, it uses similar coding patterns, such as the type system, to model the generator's control flow. This consistent way of implementing the templates across different domains contributes to the designer's learning curve and avoids inconsistencies.

Like the HW generator, the FW generator frontend consists of a set of transformation rules translating every potential CIM of an IP into an FW PIM. The transformation rules primarily describe a control flow driven by the characteristics of the IP's CIM. This way, attribute-specific fragments of the abstract firmware model are assembled into a complete firmware model. Two aspects of the generator frontend and the resulting firmware model need to be emphasized: First, the resulting PIM contains all components required to generate the ready-to-use driver. Second,

[37]Note that the CIM feeds the HW as well as the FW generator. For this reason, some attributes can be considered as hardware-only attributes that affect the resulting HW and not the driver generator, e.g., the clock *Prescaler*.

the generator builds the FW PIM in a minimalistic way exclusively tailored toward the particular CIM.



(a) *I2S CIMM* (b) *SPI CIMM*

(c) *UART CIMM* (d) *PAD CIMM*

Figure 7.2: *Examples of computational independent metamodels of communication and I/O IPs.*

Generally, each generator frontend is divided into two parts, one dealing with the structure and the other dealing with the behavioral components. Both aspects address all relevant driver parts, as shown for the UART in Listing 7.1 and 7.2. To recap Chapter 6, the fundamental entities of the FW PIMM to achieve this are objects- and action-nodes expressing the structure and behavior of the firmware.

The generator defines the driver structure by introducing objects required for the driver's operation or interface. These include macros, constants, variables, enumerated lists, types, and more. One essential entity of the structural view is the CIM-dependent device handler that specifies hardware capabilities or configurable run-time attributes. These structural entities serve as the driver's interface and objects that constitute the data and control flow. This leads to the second purpose of the generator front end, which is to construct behavior. The generator provides

transformation rules that arrange actions in a CIM-dependent sequence. For example, a sequence that describes a control function or an I/O activity.

Listing 7.1: *UART.h.*

```
/************** Include Files **************/
#define Uart_hal.h
#define System_config.h
/************ Constants & Macros ***********/
#define UART_LEN = 2;
/****************** Types ******************/
// IP specific enumerated lists
typedef enum {
   MODULE1;
   MODULE2;
   MAX_UART_LEN;
}Uart_Module_t;
// Device handler declarations
typedef struct{
   uint8_t Databits;
   uint8_t StopBits;
   uint8_t ParityBits;
} Uart_Framesetting_t;
/************ Function Prototypes ***********/
uart_init{Uart_Module_t module};
uart_transmit_blocking_uint8(
   Uart_module_t module, uint8_t data);
```

Listing 7.2: *UART.c.*

```
/************** Include Files **************/
#define Uart.h
/****************** Data ******************/
// Initialization of Variables and Handlers
Uart_Framesetting_t Uart_Module1_Framesetting{
   .Databits = 8,
   .Stopbits = 1,
   .Paritybits = 1
};
// Initialization of the configuration array
Uart_config_t uart_config[2] = {
   {<Module1 handlers>},
   {<Module2 handlers>}
};
/************** Functions **************/
uart_init{Uart_Module_t uart_module}{
   // Initialization sequence
}
uart_transmit_blocking_uint8(
   Uart_Module_t module, uint8_t data){
   // Transmit sequence
   }
```

7.1.1 Device Driver Structure

The device driver's structure defines the driver's basic skeleton, e.g., the driver interface and the data objects appearing in the driver. An essential part of the structure is the device handler. The device handler structures hardware attributes into data structures to operate driver functions in a way that keeps details behind the scene. A cleverly designed handler that follows a common standard for different variants simplifies the integration of generated drivers into the application layer. Further, it improves the portability of the generated code between different applications.

But back to the definition of a device handler. A device handler, e.g., the *Uart_Receiver_t* in Listing 7.3, specifies a data structure that describes the properties of a module. Furthermore, it is responsible for the data exchange between hardware and firmware. A handler generally includes constant hardware characteristics and features that can be configured at run-time. A handler always refers to a specific module. In contrast, a configuration table defines a handler array covering all module characteristics and features. A configuration table offers more implementation options. For example, it allows designing simplified initialization functions or batch processes. Device handlers linked in an acyclic graph are called device trees.

Inspired by the *OpenFirmware-style* of [127], the generator approach defines a configuration table with device trees. A firmware function that passes a pointer to this array can access any attribute of the module. This structure facilitates the implementation of different driver architec-

tures and increases portability and configurability.[38]

The mapping of the device tree is similar to the structure of the IP CIM. However, the generator only maps the firmware-relevant attributes in the device tree and neglects attributes that are purely relevant for hardware generation. When assembling the handler, the generator identifies those attributes across different modules. These include run-time configurable attributes and hardware capabilities that vary between modules of the IP. So, the generator tunes the device handlers depending on the IP's capabilities and attributes.

A device tree that adheres to one coding standard is easier to integrate into applications and facilitates behavioral modeling. The framework maintains a particular structure for different CIMMs, as shown in Figure 7.2, which ensures a consistent structure of the extracted device trees. For example, all CIMMs reuse the same pattern for organizing different IP modules. For example, the IP CIMMs in Figure 7.2 can contain any number of *Modules*, each managed by an individual device tree.

An important aspect of device tree generation is optimization to keep the memory footprint small. For this purpose, the generator evaluates the relevance of each attribute (is relevant if it is run-time configurable or varies between modules). Otherwise, it is omitted. Listing 7.3 applies to a particular CIM but may become invalid or inefficient for a different CIM. For example, disabling *DynamicFrame* and specifying identical frame settings for all modules would make the *Uart_Framesetting_t* handler obsolete.

Listing 7.3: *Structure of the UART source.*

```
//Uart Frame Settings.
typedef struct Uart_Framesetting_t{
    const uint8_t Databits; /**< Number of transferred data-bits */
    const uint8_t StopBits; /**< Number of transferred stop-bits */
    const uint8_t ParityBits; /**< Number of transferred parity-bits */
}Uart_Framesetting_t;

//The receiver handler.
typedef struct Uart_Receiver_t {
    struct Fifo_t FIFO; /**< Handler for fifo pattern */
    Status_t Status; /**< Receiver status */
} Uart_Receiver_t;

//The transmitter handler.
typedef struct Uart_Transmitter_t {
    struct Fifo_t FIFO; /**< Handler for fifo pattern */
    Status_t Status; /**< Transmitter status */
} Uart_Transmitter_t;

//The root structure of the Uart device tree
typedef struct Uart_Config_t {
    uint16_t * Prescaler ; /**< Handler for data transmission baudrate */
    struct Uart_Framesetting_t * FrameSetting; /**< Protocol frame settings */
    struct Uart_Receiver_t * Receiver; /**< Module has Receiver */
    struct Uart_Transmitter_t * Transmitter; /**< Module has Transmitter */
} Uart_Config_t;
```

[38] A pointer array can result in additional overhead being required to dereference the pointer [127].

7.1.2 Device Driver Behaviour

Besides the structural components, the generator frontend template contains transformation rules for creating the driver behavior. Once again, the transformation creates no general-purpose behavior but a fully functional behavior tailored to the specification. Thus, this generator addresses typical drawbacks of FW generator approaches, summarized in Chapter 3. The framework provides features to specify the control and data flow and a flexible modeling language to describe the device driver behavior in detail.

In order to describe a function or an activity as a transformation rule, the designer specifies the function's requirements. These requirements are satisfied when generating the function. For example, a UART- or I2S-transmit function cannot be built unless the CIM covers a module with a transmitter. Furthermore, the designer has to identify all attributes of the CIM which influence the function's behavior. These attributes impact the generator's control flow. This control flow is essential as it ensures diversity and adaptability to the CIM and, thus, to the hardware of the IP. The generator's designer provides for each generator's control node a certain sub-behavior. Each sub-behavior is defined as a group of actions which is also considered a basic block. Once the generator is executed, it assembles the sub-behaviors into an overall behavior, the final activity.

Listing 7.4: *The transformation rule inside the SPI generator frontend that generates the spi_control function.*

```
1  def ActionControl(self):
2      activity = self.addfunction(Name="spi_control", Type="bool")
3      group = activity.createGroup()
4
5      master_group = group.getIfGroup(self.Master)
6      if master_group[0]:
7          transmission_group = master_group[0].getIfGroup(self.Master & self.DynamicFrame)
8          if transmission_group[0]:
9              transmission_group[0].addWrite(Name=BF.POLARITY, Value=self.Polarity)
10             ...
11             transmission_group[0].addWrite(Name=BF.HOLD_CS, Value=self.HoldCS)
12
13         prescaler_group = master_group[0].getIfGroup(self.Master & self.DynamicPrescaler)
14         if prescaler_group[0]:
15             prescaler_group[0].addWrite(Name=BF.PRESCALER, Value=self.Prescaler)
16
17         slave_sel_group = master_group[0].getIfGroup(self.Master & self.DynamicSlaveSel)
18         if slave_sel_group[0]:
19             slave_sel_group[0].addWrite(Name=BF.SLAVE_SEL, Value=self.SlaveSel)
20
21         data_width_group = master_group[0].getIfGroup(self.Master & self.DynamicWidth)
22         if data_width_group[0]:
23             data_width_group[0].addWrite(Name=BF.WIDTH, Value=self.DataWidth)
24
25     group.addReturn(Name="True")
```

Listing 7.4 shows a template that instantiates an activity of the FW PIM. The activity describes the *spi_ control* function, configuring and initializing a SPI module at run-time according to the parameters stored in the associated device handler. It includes several action groups for setting different bit fields. Lines 8-10 of the template generate the code for configuring the transmission settings, line 14 the Prescaler, line 18 the slave-select and line 22 the data-width. However, each action group is only generated for a specific set of CIM attributes. For example, an SPI CIM that feeds the generator template may result in the target code of Listing 7.5.

The FW template language provides a conditional generator node defined as `getIfGroup` with an argument for a generation condition. When instantiating the activity, the generator distinguishes between the following three scenarios at these nodes.

- No module supports the generation condition. Consequently, the whole activity group is not created. For example, the SPI CIM that results in the provided target code contains no module being a master with run-time configurable slave selection. Accordingly, the generation condition in line 16 of the template is not fulfilled, and the associated action group is not created.
- Not every module supports the generation condition, but at least one module does. In this case, the action group is enclosed with a conditional statement. For example, this applies to the transmission settings in lines 8-12 of Listing 7.5 since some modules provide no hardware support to configure the transmission settings dynamically.
- All modules support the generation condition. So, no conditional statement is required. The associated activity group is added to the group (see data width and Prescaler configuration in lines 14-15 of Listing 7.5).

Listing 7.5: *A potential spi_ control function generated for a particular SPI CIM.*

```
1  /**
2   * Control function to configure the SPI module.
3   * @param modIdx defines the selected modules via Enumeration index
4  **/
5  bool spi_control(SPI_Enum modIdx){
6      SPI_Config_t *module; /**< Load device handler.*/
7      module = &SPI_Config[modIdx];
8      if (SPI_Config.Master_Config->TransmissionSettings){
9          SPI_HAL[modIdx].POLARITY_WRITE(SPI_Config.Master->TransmissionSettings->polarity);
10         ...
11         SPI_HAL[modIdx].HOLD_CS_WRITE(SPI_Config.Master->TransmissionSettings->hold_CS);
12     };
13
14     SPI_HAL[modIdx].PRESCALER_WRITE(SPI_Config.Master->Prescaler);
15     SPI_HAL[modIdx].WIDTH_WRITE(SPI_Config.Master->DataWidth);
16     return True;
17 }
```

The template of *spi_ control* demonstrates that the template language is abstracted from the target languages. However, the level of abstraction remains very close to the target languages.

For example, the entities (e.g., variable widths) are described more abstractly, while the behavior is specified in detail. This facilitates the best possible use of the language while maintaining a wide range of variants without implementing platform-specific target code directly. This diversity is already evident in this simple spi_control function. Moreover, the coding effort for the shown template is comparable with the generated code. With increasing function complexity, the coding effort for the template may increase. However, the framework keeps the design effort low since it provides a FW DSL and a simplified interface for design pattern reuse.

7.2 Domain Specific Language and Design Pattern Reuse

The auto-generated API of the FW-PIM is used to specify the transformation rules of the embedded software template. However, mapping between IP CIM and FW PIM solely based on the API requires a non-negligible amount of effort when implementing the template since each model feature must be instantiated individually. A benefit of the proposed methodology is reducing the effort by including an advanced FW DSL and design patterns.

The FW DSL generally performs tasks that go beyond the simple standard API derived from the metamodel. A DSL function specifies a sequence of API calls to construct more complex patterns of the underlying metamodel, as shown in 7.1. Furthermore, as described in [85], the FW-DSL simplifies the construction of the PIM by providing more object-flow-driven programming constructs. In addition, the DSL combines several aspects of the abstract firmware model in a single function. In this way, more advanced modeling sequences can be performed in a single call, e.g., conditional assignments, data transformations, and loop-based initialization.

A firmware design pattern implements a sub-behavior of the driver as a sub-template. Like the generator frontend, a sub-template provides transformation rules that can be included in different IP driver templates[39] or functions. A design pattern mainly implements recurring tasks, such as initialization sequences, status monitoring, or IP controls.

Both the DSL and the design patterns reduce template design effort, minimize coding errors, and increase consistency between IP drivers. Note that a designer can customize the FW DSL by configuration and keeps the freedom of introducing new design patterns.

7.2.1 Domain Specific Language

Automatically generated APIs, such as those provided in this framework, define standard building blocks for instantiating and analyzing classes or attributes of a metamodel. An API provides for each class and property access method. When writing a template solely with API functions, each object and attribute must be treated individually. The DSL is considered an API extension that specifies domain-specific building blocks. In [85], an automated approach is provided that generates an expressive python-based DSL by combining building blocks of the API into domain-specific routines[40].

[39] A firmware design pattern is only suitable for different IPs if they are subject to the same HW. The best way to ensure portability is to use a corresponding HW design pattern for the HW generator as a counterpart to the FW pattern. An example of an HW and FW design pattern is the register interface, which is reused in various IPs.

[40] The DSL automation approach can be applied to different domains. In addition to the FW generation framework, it has been applied, for example, to the HW modeling framework and the formal verification framework.

The firmware framework uses the approach of Han et al., which relies on a metamodel that includes all API functions. An instance of this metamodel employs the metadata of a target domain, e.g., the FW PIMM. It maps all entries of the metamodel and their API functions. Within this model, the designer can assemble API functions into more advanced assignment sequences. These constitute the source of the DSL, which is generated in an automatic process. The example in Listing 7.6 demonstrates the use and capabilities of different FW DSL routines to construct a square root function template.

Listing 7.6: *A square root function template*

```
1  function = sActivity(Name="square_root", ObjProps(Type="float") // "Babylonian method"
2  function.value = sVariable(ObjProps=ObjProps(Type="float") // "is the radicand."
3  group = function.createGroup()
4  group.y = sVariable(Name="y", ObjProps = function.getOptProps()); // "square root result."
5
6  // value >= 0
7  preCondition = sCondition(Ins = [function.value, 0], Operator = ">=")
8  // y*y == value
9  postCondition = sCondition(Ins = [y * y, function.value], Operator = "==")
10
11 sub_group, post_group = group.addContract(In=preCondition, Out=postCondition)
12 sub_group.addAction(...) // "Approximating the square root through Babylonian method."
```

Object-oriented languages such as Python allow polymorphism and, accordingly, operator overloading. Operator overloading reduces coding overhead and moves potentially complex API routines into an object method that is called using standard operator syntax. lines 2 and 4 demonstrate the use of a DSL function for simplified instantiation of a *Variable* as an FW PIM object. The standard way of instantiating the variable is to use the API function `createVariable` (generally: `create<ClassName>`) and then use other API calls to set the variable's properties. Instead of utilizing the API, the DSL overloads the built-in `__setattr__` routine to invoke the `createVariable` function under the hood and to set all properties. Thus, the designer can instantiate the object by a simple object assignment.

Similarly, the designer defines comments in the template assigned to the target code, as shown in lines 1,2,4 and 12. However, for commenting, the DSL overloads the "//" operator. In other words, the `__floordiv__` function is overloaded to call the API function `setDescriptions()`, as shown in Listing 7.7. Likewise, the multiplication `__mul__` or other arithmetic expressions are overloaded in order to create PIM internal expression nodes. An application is shown in line 9 (see `y * y`) of the square_root template. The listed example depicts a more object-flow-oriented programming approach of the DSL.

Listing 7.7: *The FW DSL with operator overloading for the variable class.*

```
1  class sVariable:
2      def __floordiv__(self, other):
3          self.setDescriptions(other)
4
5      def __mul__(self, other):
6          self.addOperation(Operator="+", Ins[self, other])
```

A more concrete example of a DSL function is given by `addContract()` in line 11. A contract describes a run-time check through a pre-condition, post-condition, or both of a sub-routine. The DSL specifies a contract as an entity of an *ActivityGroup* manifested by one or two conditions (see lines 6-9). The pre-condition of the square root is that an argument is a natural number. As a post-condition, the square of the calculated result must be equal to the argument. This behavior is achieved by the DSL that invokes a sequence consisting of an if-statement (in FW PIM: Decision) enclosing the contracted sub-routine, followed by another if-statement. The result of the discussed template using the C-generator is shown in Listing 7.8.

Listing 7.8: *The generated C target code obtained from the square root template in Listing 7.6.*

```
1  /**
2  * Babylon method
3  * @param value is the radicand.
4  * @return square root result.
5  **/
6  float square_root(float value){
7      float y = 0; // square root result.
8      // Contract: Pre-condition
9      if (value >= 0 ){
10         // Approximating the square root through Babylonian method.
11         ...
12     }
13      //Contract: Post-condition
14     if (y * y == value) {
15         return value;
16     }
17     return -1;
18 }
```

Further on, Han et al. [86] introduce the self-verifying DSL, extending the DSL generation framework with automated complementary tests. These tests describe various test inputs to verify the DSL functionalities and thus assure the quality of the DSL, such as the FW DSL.

7.2.2 Design Pattern

Design patterns in the FW modeling framework are well-established generator templates for solving recurring design problems across different IPs. Like the generator frontend, a design pattern depicts transformation rules for problem-solving that can be used in a specific context, such as communication IPs. Accordingly, a design pattern is implemented through API and DSL functions. Compared to a DSL routine, a design pattern can be further populated by a sub-structure of the IP CIM. Note that all IPs applying the design pattern must share the same sub-structure in the CIM, as is the case for the FIFO structure (in UART, SPI and I2S) in Figure 7.2.

The FIFO pattern describes alternative generation flows as subject to the CIM configuration, as discussed in Section 7.1.2. Depending on whether the HW module includes a FIFO logic or provides a simple register buffer, the generator constructs the correct code for reading. There are two possible solutions: Either it is resolved as simple register access to read the single data register *RX_DATA* or as a loop of accesses to read the whole FIFO. Listing 7.9 shows the generator that

is able to construct the FIFO design pattern depending on the availability of the FIFO in the HW module.

There are no limits to the complexity of the design patterns. The only limitation to each design pattern is that they must construct a valid inherent sub-structure of the PIMM, e.g., a partial behavior of the driver function. The use of design patterns in various generators is recommended since design patterns are more robust through continual improvements.

Listing 7.9: *The design pattern for reading a FIFO or buffer.*

```
def FifoBufferRead(self, Group=None, SetVar=None, ReturnVar=None, Style=None):
    """
    :param Group: The current activity group which is extended by the FIFO/buffer.
    :param SetVar: The variable that stores the read data.
    :param ReturnVar: Return value in case something went wrong (NonBlocking)
    """
    # Buffer space is available?
    fifo_group = Group.getIfGroup(Name=[self.FifoReceiver, None])
    if fifo_group[0]:
        fifo_group[0].addBlocking(Name=BF.RX_EMPTY, Style=Style, Return=ReturnVar)
    if fifo_group[1]:
        fifo_group[1].addBlocking(Not=True, Style=Style, Name=BF.DATA_AVAIL, Return=ReturnVar)
    # Read Data from FIFO
    Group.addWrite(Name=SetVar, Value=BF.RX_DATA) // "Read data from buffer"
    # Send notification that data was read.
    fifo_group = Group.getIfGroup(Name=[self.FifoReceiver, None])
    if fifo_group[0]:
        fifo_group[0].addWrite(Name=BF.BYTE_READ, Value=1) // "FIFO: Data is available?"
    if fifo_group[1]:
        fifo_group[1].addClear(Name=BF.DATA_AVAIL) // "Buffer: Data is available?"
```

Chapter 8

Device Generic Generator Backend

The scope of device driver variants that the generator frontend can construct depends on the number of IP requirement combinations. However, the framework increases flexibility through the generator backend, providing further customizable cross-IP transformations on the abstract firmware model. This generator backend phase is IP-independent. It performs endogenous transformations to realize various design decisions, such as software architectures for a given IP. So the development effort reduces as these transformations are implemented in a one-time effort and can be reused for all peripheral components. Thus, the generator remains hidden from the developer of a new IP.

The generator backend performs horizontal transformations that do not affect the essential characteristics of the assembled abstract firmware model. Consequently, the underlying core behavior is unchanged. Some of these transformations incorporate additional features into the abstract design, e.g., safety patterns or debugging nodes. Others are intended to realize various firmware design and architecture decisions. Note that all these transformations are driven by the generator-spec metamodel discussed in Section 6.3.

Another task of the backend is the translation of the PIM into the PSM. For this purpose, the abstract firmware constructs are mapped to the chosen language model, e.g., C, Rust or `C++`, which feeds the code generator described in Chapter 4. The framework extends this step with a subsequent compilation step that builds the binary. For this, the backend feeds the generated code into a customizable compiler flow, including a range of optimization flags based on GCC. Due to the transformations on the platform-independent level, the different language generators and the configurable compiler flow, the backend can create a large number of IP variants that form the design space.

A designer benefits from model-based development by selecting a target instance from this design space. In addition, it enables trade-off analysis [166] that assists the designer in filtering suitable variants of that space. A trade-off analysis in an embedded context is about determining the impact of different factors on various design objectives (costs) such as performance, code segment size (.text), data segment size (.data), and fault coverage. Note that no design decision simultaneously optimizes all objectives. Instead, it deteriorates one cost parameter while improving another. Conclusively, the trade-off analysis based on the Pareto principle[41] provides variants

[41] The Pareto principle addresses the multi-objective optimization problem and identifies a set of non-dominated solutions. A solution is considered optimal if no objective can be improved without degrading other objectives [139].

that meet the system's non-functional cost requirements [124, 211].

This chapter details the backend and introduces two examples of trade-off analyses. One deals with optimizing compiler flags, the other with register interface optimization.

8.1 Design Decisions

Design decisions, such as the Device Driver Architecture and Design, are defined in the generator-spec introduced in 6.3.2.1. All design decisions are legitimate and can be applied to any existing or future IP. For example, the designer can define the driver architecture of the IP as either generic or module-specific. When choosing the generic implementation, the generator creates a generic function, while a specific implementation is tailored to each module. Moreover, a designer can define the IP design to realize a driver function following an interrupt-, blocking- or non-blocking-based design. As approved by Yang et al., each of these settings has its advantages. Their work in [228] gives empirical evidence for and against polling[42]- and interrupt-driven I/O in terms of performance. The paper in [169] provides similar results, pointing out the advantages and disadvantages of purely synchronous RTOS-based and interrupt-based designs.

Overall, these driver configuration options and the IP configuration lead to an enormous number of driver variants. Design exploration becomes possible since the backend automatically executes the architecture and design transformations. This capability is essential since each design decision is a legitimate choice that significantly impacts performance and memory requirements.

8.1.1 Driver Reusability

A designer's choice in the generator-spec is whether the driver architecture follows a generic or specific implementation. A generic driver design considers the properties of all modules of a device in a single driver. So, it provides a driver reusable for various modules of the IP. In contrast, in a module-specific design, the generator tailors the driver design to the characteristics of a single module. Accordingly, each module is specified by a unique set of functions.

The pseudo-code in Algorithm 3 shows a snippet of the `spi_receive` function dependent on the generator-spec. The example demonstrates that a generic architecture results in a higher number of branches and control blocks, which slows down the performance of the design. However, a module-specific solution omits those control blocks, such as the termination conditions in lines 7-10. In particular, a generic function may inflate due to hardware-dependent alternative control flows, as shown in lines 13-17. This increasing complexity is especially prevalent whenever there are many hardware dissimilarities between the modules.

Furthermore, the bitfield accesses, as well as the function declaration, adapt to the architecture style. Both are optimized in the module-specific implementation and are more performant using direct accesses and less de-referencing. For example, a generic variant can only be executed with a device handler de-referenced in advance (lines 2-4). This handler determines the control flow of the module through the device driver. A handler may become obsolete in a module-specific implementation since it does not need to capture the hardware settings to determine the control

[42] Polling is another wording for blocking. It specifies a synchronous I/O realized by a busy-wait loop that keeps the CPU busy while waiting for completion.

Algorithm 3: Pseudocode of a blocking SPI receive function as generic or module-specific variant.

```
   /* Function declaration                                                    */
 1 if generator-spec == Generic then
 2 |   SPI_RESULT receive_spi(SPI_Enum modIdx, uint16_t len, uint32_t *rxbuf){
 3 |       SPI_Config_t *spi; /**< Load configuration from the modules config list*/
 4 |       spi = SPI_Config[modIdx];
 5 else
 6 |   SPI_RESULT receive_spi_module1(uint16_t len, uint32_t *rxbuf){
   /* Leave Condition:  Only generated if at least one module meets the condition.  */
 7 if generator-spec == Generic then
 8 |   if(( spi.Master & !spi.MISO ) | ( spi.Slave & !spi.MOSI )){
 9 |       return NO_HW_SUPPORT;
10 |   }
11 while(len){
   /* FIFO or Buffer available?                                               */
12 if generator-spec == Generic then
13 |   if (spi.FIFO){
14 |       while (SPI_HAL[modIdx].RX_EMPTY_READ()){};
15 |   } else {
16 |       while (SPI_HAL[modIdx].RX_WORD_READ()){};
17 |   }
18 else
19 |   while (SPI_HAL[module1].RX_EMPTY_READ()){};}
   /* Read data.                                                              */
20 if generator-spec == Generic then
21 |   uint32_t cache = SPI_HAL[modIdx].DATA_RX_READ();
22 else
23 |   uint32_t cache = SPI_HAL[module1].DATA_RX_READ();
24 ...
25 }
```

path.

As noted, the control flow of a generic architecture considers all possible modules of one IP and therefore differs in complexity. In contrast, the module-specific variant is tailored to the respective instance. So, depending on the IP CIM, either the generic or module-specific architecture can be beneficial. Table 8.1 summarizes the data for different device driver architectures, such as generic device drivers to handle multiple modules in one function or module-specific driver functions. The average number of instructions for each function of a generic driver is significantly larger than the number of instructions in the specific driver. There are several reasons: First, the function includes more conditional branches and control blocks to handle the different modules. Second, the device handler and hardware registers are accessed generically. So, the choice of architecture is a trade-off between performance, memory footprint, and program size.

On average, each device driver requires 55% more instructions per function for four randomly assembled modules when the generic design is chosen over a specific design. Also, this implies deviations in performance (number of clock cycles). Contrary, the generic design reduces the .bin size by an average of 11% compared to the specific design. However, with an increasing

similarity of the modules, fewer control blocks are in the generic design. This results in a minor performance deviation (-36%) and higher memory footprint reduction (24%) between generic and specific designs. As highlighted in Table 8.1, a generic solution should be contemplated if the configurations of the modules within the IP differ only slightly.

Table 8.1: *Comparison of different device driver implementations using no compiler optimization.*

	4 random modules				4 identical modules			
	Generic driver		Specific driver		Generic driver		Specific driver	
Peripheral	.bin size	functions / instructions Ø	.bin size	functions / instructions Ø	.bin size	functions / instructions Ø	.bin size	functions / instructions Ø
UART	8373	(8) 78,9	8781	(28) 26,4	10448	(8) 57,3	12744	(32) 32,3
DMX	7626	(8) 93	8734	(26) 39,0	7080	(8) 62,4	10156	(32) 39,4
I2S	8296	(7) 102,3	7740	(20) 29,2	5924	(5) 45,6	7300	(20) 28,6
SPI	8556	(11) 71,2	8804	(30) 27,9	10936	(9) 51,7	13540	(36) 31,4
PAD	3464	(7) 52,3	3968	(18) 27,7	2268	(3) 42	3108	(12) 28
Timer	13324	(13) 57,1	18836	(52) 40,7	12524	(13) 57,1	18036	(52) 40,8

8.1.2 I/O Driver Design

The type of synchronization for each I/O module can be chosen from a set of alternatives. As described in Section 6.3.2.1, a distinction is made between blocking, non-blocking and interrupt-based I/Os. A blocking driver is the easiest to implement, as it requires no additional hardware and no additional synchronization at the application layer. Instead, a blocking-based driver is synchronized by a synchronization or polling flag in the driver function.

A non-blocking process, as the name implies, does not block the CPU from executing other processes. Instead, a non-blocking driver immediately returns from the function if the condition for continuing the process is not met. The biggest challenge and the most important aspect of a non-blocking I/O function is the task of maintaining the function status. So the operating system or application layer can thus continue executing the process from the previously stored state. This design requires no additional hardware but further attributes in the device handler to store the state.

When the system includes a programmable interrupt controller HW, an interrupt-based device driver can be implemented. In addition, the IP must supply interrupt wires that the interrupt controller gathers. An interrupt signal from an I/O IP is set when the IP hardware triggers a specific event, e.g., a task is completed, or an error occurs. The I/O module reports the interrupt through this signal, which causes the processor to stop the current process and to execute the exception handler. The exception handler determines the cause of the asynchronous or synchronous exception and invokes the associated interrupt service routine (ISR). The ISR processes the operation for the module that triggered the interrupt. After returning from the routine, the CPU restores the system's state before being interrupted.

The main benefit of the interrupt-based design is that the CPU can work continuously on a task without permanently checking the I/O devices. The IPs themselves can interrupt the CPU

when necessary, which might prove to be more efficient. When using such a design, it is important to keep two aspects in mind. First, an ISR should be short and compact since an interrupt service routine prevents other lower-level interrupts from occurring. Second, an interrupt-based design adds additional complexity to hardware and software, e.g., event handling, context switches or interrupt nesting. This additional complexity defines the interrupt latency, the delay between triggering the interrupt and executing the first instruction of the ISR. The framework offers various settings for configuring the complexity and latency of the interrupt-driven design. In the following subsections, common choices are presented, e.g., vectorized or non-vectorized.

Table 8.2: *Evaluation of coding effort for different driver designs. (blocking, non-blocking and interrupt*

Peripheral	Generated device driver (SLOC)			Template (SLOC)
	interrupt	non-blocking	blocking	
UART	232	236	226	302
DMX	192	196	187	377
I2S	312	321	303	489
SPI	360	364	348	471
PAD	430	430	422	440

A comparison between the three supported variants by source lines of code (SLOCs) is provided in Table 8.2. A blocking driver generally requires fewer resources but can become inefficient for poor throughput. An interrupt-based routine is usually faster but can become complex as the number of IPs working together increases. Ultimately, the design decision must be made considering all application requirements (e.g., time constraints and functional safety requirements). However, in many applications, a blocking driver is sufficient.

8.1.2.1 Synchronous Driver

The generation of the different driver designs is done by endogenous transformations. The main idea is to identify the parts that differ between the designs and offer alternative transformations based on the design decision. These parts are defined in the abstract model by artifacts resolved differently in the generator backend.

The pseudocode in Algorithm 4 points out these parts and shows the resulting code generated depending on the generator-spec. The behavior of the `receive_uart` function between the non-blocking and blocking variants is quite similar. However, depending on the design choice, the two differ in a single control node labeled to be resolved. As shown in line 11, the blocking variant resolves this node as a waiting queue used to synchronize the driver. A while loop stops the process until the exception conditions (e.g., TX_FULL) are met. In contrast, the non-blocking variant resolves the node into a simple conditional statement, as shown in lines 13-15. The function is aborted, and the intermediate result is returned unless the operable condition is met. Therefore, the higher software layers must proactively call the function again later to complete the process. This behavior differs from asynchronous design, which uses an interrupt to invoke a function if the device fully satisfies the operating condition.

Algorithm 4: Pseudocode of the generated receive function for different driver architectures in a module-specific implementation.

```
   Input: Type = Device driver architecture [blocking, nonblocking, interrupt]
 1 receive_uart_<Type>(
 2 if not Type==interrupt then
 3 |   uint8_t* data, uint16_t len                    // Function arguments for non-interrupt
 4 )
   /* Arguments can not be passed within interrupt service routine.                  */
 5 if Type==interrupt then
 6 |   interrupt_pre_function();                   // Interrupt controller dependent actions
 7 |   uint8t data = rx_data_uart;
 8 |   uint16t len = FIFO_SIZE_UART;
   /* Core behavior of receive function                                              */
 9 while (length > 0){
10 if Type==blocking then
11 |   while(bitfield_get(RX_EMPTY_UART));                                  // Busy-loop
12 else if Type==nonblocking then
13 |   if(bitfield_get(RX_EMPTY_UART)){                                 // Process is queued
14 |       return len;
15 |   }
16 *data = bitfield_get(RX_DATA_UART);
17 bitfield_set(BYTE_READ_UART);
18 *data++;
19 len=len-1;
20 }
   /* Return from function                                                           */
21 if Type == interrupt then
22 |   interrupt_post_function();                  // Interruptcontroller dependent actions
23 |   return;
24 else
25 |   return len;
```

8.1.2.2 Asynchronous Driver - Interrupt Service Routine

According to the RISC-V Privileged Architecture Specification [16], the term exception refers to an unusual condition that occurs synchronously at run-time with executing an instruction in the RISC-V CPU core. An exception that occurs asynchronously to regular program execution and disrupts the control flow is called an interrupt. In other words, an interrupt is an event caused by a component other than the CPU. Note that the term exception is used for asynchronous and synchronous events in the following.

The software consists of two parts in order to handle exceptions. First, an exception handler, also called a trap handler, determines how the program behaves in case of an exception. Second, an ISR serves the interrupt. There are different modes for handling interrupts and exceptions. Through the MDA approach and configurable CIMMs, different variants of the exception unit and a fitting interrupt controller can be defined and generated. Depending on the configuration, this may also affect the I/O driver functions. The general exception handling flow according to the RISC-V specifications is outlined before describing these two key hardware components and

enumerating their configurable modes. Figure 8.1 shows the different phases processed from an exception's occurrence until completion.

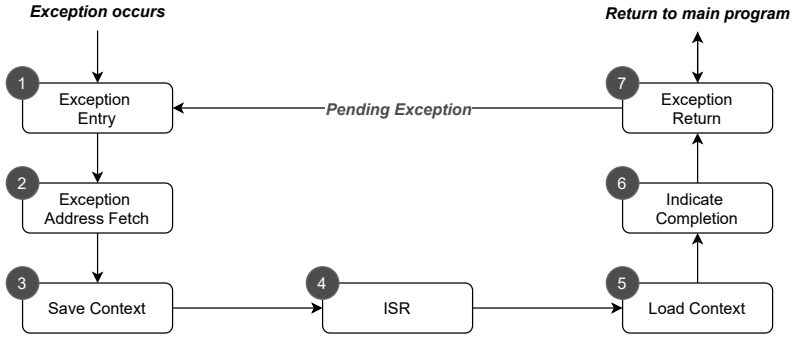


Figure 8.1: *The phases of exception handling according to the RISC-V standard.*

1. The processor receives a request to handle an exception. The request is prioritized depending on the appearance of other exceptions with higher priority and the CPU status.
2. Once the request is accepted, the processor disables the interrupts and invokes the trap handler, which handles the requested exception.
3. Upon entering the handler, the processor must save its state by context switching. The state information to be stored includes all relevant resources used by the interrupted program that the exception handling routine may also use.
4. The processor services the exception by executing the interrupt service routine assigned to the exception.
5. Once completed, the processor returns to the regular program running when the exception occurred. Thus, the program's context is restored.
6. A `mret` instruction[43] is executed after handling the exception to indicate the completion. This instruction notifies the processor to re-enable the interrupts to receive new requests from the interrupt controller.
7. Before returning, the processor checks whether exceptions are pending. If this is the case, the exceptions are chained, and the pending exception is serviced. Otherwise, the processor continues with the regular program.

These phases may vary depending on the configuration of the interrupt controller and the exception unit. The most significant types of exception handling are discussed in the following. However, first, the interrupt controller and the exception unit are introduced.

Exception unit and interrupt controller The hardware components required for exception handling in a RISC-V SoC are the Exception Unit (EU) and the Programmable Interrupt Controller (PIC). The EU is the central control unit for handling exceptions and is implemented inside

[43]The RISC-V specification defines specific instructions for returning from a trap handler. Each privilege level has its own trap handler return instruction: MRET (machine mode), SRET (supervisor mode), and URET (user mode). The framework's generated SoCs only require the machine mode trap handler return (mret).

the instruction fetch stage of the processor's pipeline. This position allows the EU to set the program counter to the address of the prioritized exception handler. Besides, it can stall or flush the CPU pipeline and directly modify specific CSRs related to exception handling (e.g., mstatus, mtvec, mepc). For this reason, the EU is configured and maintained in the CIM of the RISC-V core.

The second component, the PIC, is implemented outside the CPU core and is responsible for handling external interrupt events. The unit collects and manages all available interrupt sources from different peripherals of the SoC. Its main task is to process all those and forward the relevant interrupt request to the CPU. Next, the CPU accepts the interrupt when no other exception with higher priority is currently running or stacked. Otherwise, the interrupt waits until all stacked higher priority exceptions are processed.

The PIC created by the framework is specified by a highly configurable PIC-CIMM shown in Figure 8.2, providing many variations. Like other I/O IPs, its instances (PIC CIMs) also feed a dedicated HW and a FW generator that constructs both HW and FW, depending on the specification. The designer can configure the general behavior of the interrupt controller and each IRQ (interrupt request).

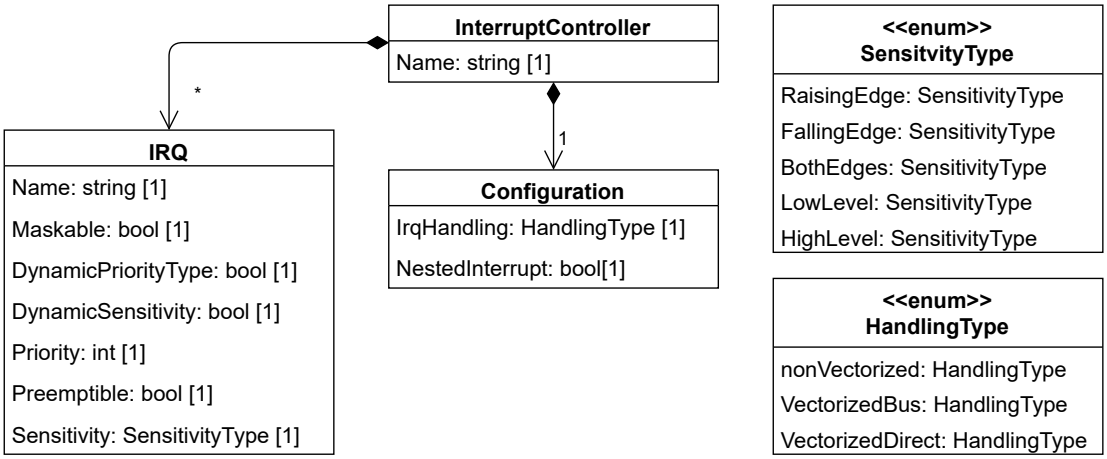
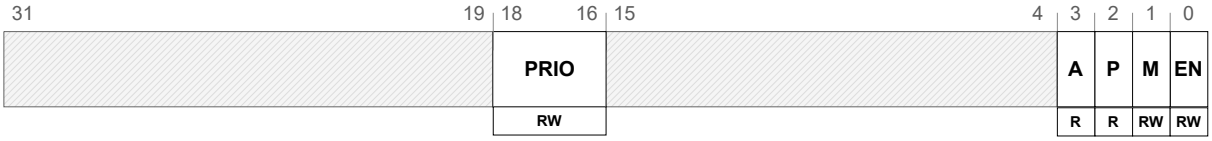


Figure 8.2: *The CIMM of the interrupt controller.*

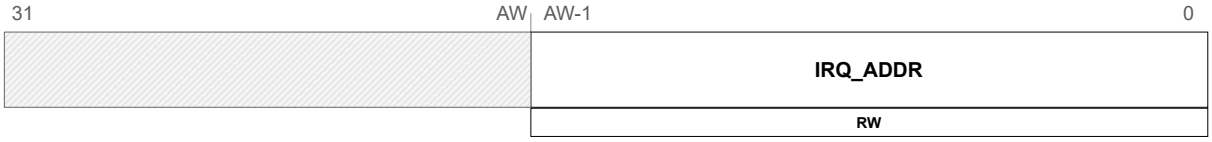
A *IRQ* is triggered by a signal event that defines edge-based or level-based *Sensitivity*. *DynamicSensitivity* can change this sensitivity via a bit field at run-time. Furthermore, each IRQ can be defined as *Maskable* and *Preemptible*. A maskable hardware interrupt can be disabled via CPU instruction, while a non-maskable interrupt is always enabled and never ignored. Meanwhile, a preemptible interrupt defines a critical task that another IRQ cannot disrupt. Besides, a *Priority* is assigned to each IRQ to determine the order between different IRQs. Also, the priority level is configurable via a bit field when activating *DynamicPriority*.

Further, two attributes are included to shape the essential behavior of the PIC independent of the specific interrupt channels. One feature is the *IrqHandling* which influences the exception address fetch stage of the exception handler (see Figure 8.1). Second, the choice of *NestedInterrupt* determines whether the interrupt controller can handle the preemption and nesting of interrupts inside its core. The impact and trade-off of both features on the system are elaborated in the following paragraphs.

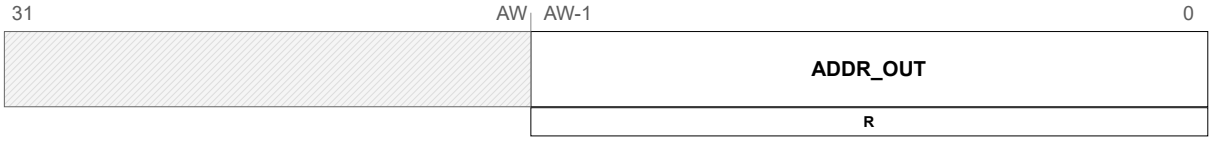
The PIC includes a register interface, introduced in Section 5, assembled depending on the PIC configuration. Figure 8.3 shows the generated registers addressed by the firmware to control the interrupt device.



(a) *IRQx_CTRL*



(b) *IRQx_ADD*



(c) *ADDR_OUT*

Bitfield	Size	Type	Description
EN	1	RW	**IRQ source enable** 0 DIS: The IRQ channel X is disabled. 1 EN: The IRQ channel X is enabled.
M	1	RW	**Mask interrupt channel** 1 MASK: The interrupt channel X is masked. 0: UNMASK: The interrupt channel X is unmasked.
P	1	R	**Pending interrupt** 0 RESOLVED: The interrupt X is resolved. 1 PENDING: The interrupt X is taken and has not yet been finished
A	1	R	**Active interrupt** 0 INACTIVE: The interrupt X is not executed. 1 ACTIVE: The ISR of interrupt X is currently executed.
PRIO	3	RW	**Priority** The priority level of the interrupt X.
IRQ_ADDR	AW	RW	**Interrupt jump address** Stores the address of the ISR associated with interrupt X.
ADDR_OUT	AW	R	**Interrupt active address** Stores the address of the ISR associated with the active interrupt.

Figure 8.3: *Definition of the register layout of the interrupt controller specified within the register interface model discussed in Section 5.*

Vectorized and non-vectorized interrupts The PIC offers three types of interrupt handling, defined by the PIC CIM. These modes are grouped as non-vectorized or vectorized, while vectorized is further classified as vectorized-bus or vectorized-direct. The generated firmware required to resolve an interrupt for each handling mode is shown in Figure 8.4.

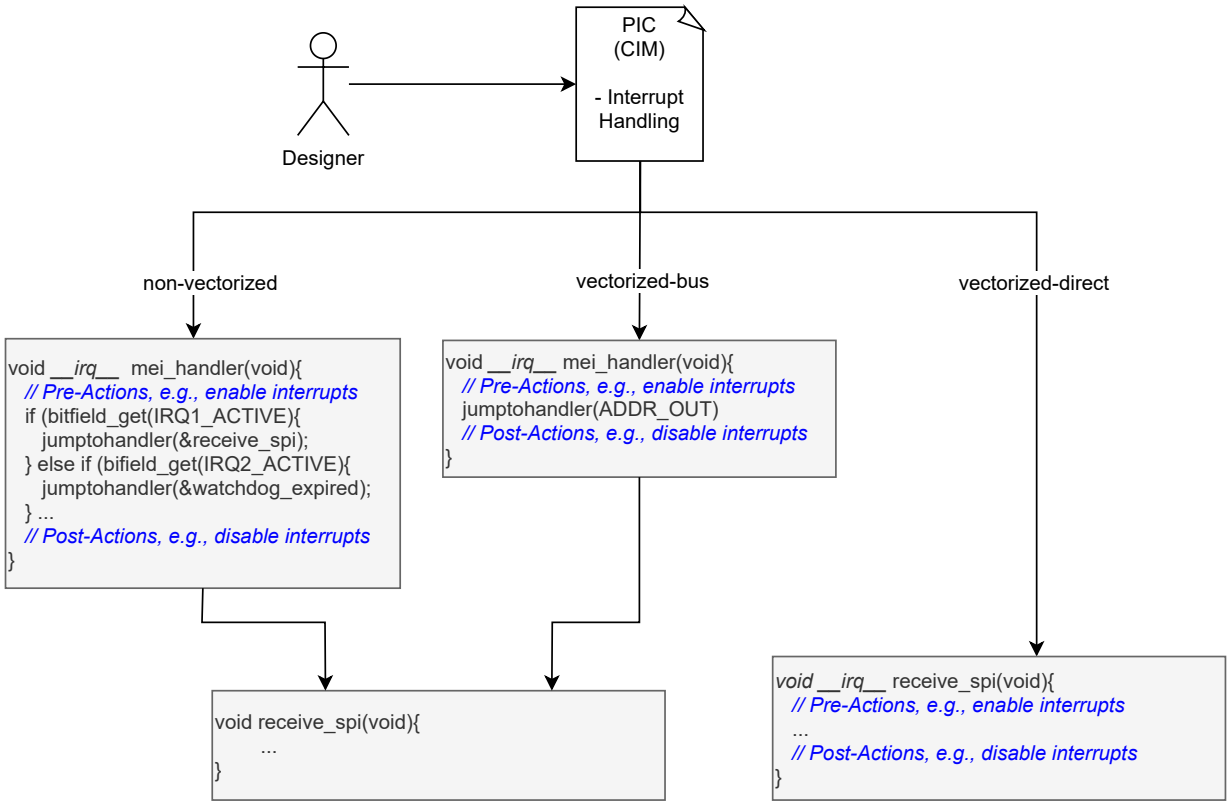


Figure 8.4: *Interrupt handler generation flow.*

In the non-vectorized interrupt handling, the PIC jumps to the same destination address (vector entry) for each interrupt source. The firmware defines the location of the vector entry in the mtvec CSR[44] during the initialization phase of the PIC. In general, this destination address specifies the location of the interrupt handler that contains the interrupt vector table. The interrupt handler resolves the active interrupt source by looping through all available interrupt sources of the table and invokes the corresponding service routine. Compared to a vectorized PIC, a non-vectorized PIC entirely shifts the task of identifying and invoking the correct interrupt address to the software domain.

In vectorized-bus mode, the interrupt address is resolved within the PIC, not by firmware. In this mode, the PIC generator creates for each interrupt source IRQx an additional register called IRQx_ADDR, which specifies the destination address of the interrupt service routine. Thus, these registers populated by firmware instructions constitute the interrupt vector table. When an interrupt occurs, the hardware resolves the interrupt request and writes the associated ISR address from the vector table to the ADDR_OUT register. Like the non-vectored mode, the CPU again triggers a jump to the interrupt handler for each interrupt. However, the interrupt handler's complexity is reduced since it only needs to retrieve the address from the ADDR_OUT register to invoke the specific ISR. So, there is no need to loop through all the different interrupt sources anymore.

The vectorized-direct mode offers the fastest concept as it delegates the whole handling of the specific interrupt service routine from the firmware to the hardware. The address is resolved within

[44]The machine trap-vector base-address (mtvec) stores the destination address that overwrites the program counter in case of an exception.

the PIC and propagated to the CPU. For this purpose, the ADDR_OUT register is replaced by a direct connection to the CPU. So, the address of the interrupt service routine is calculated and propagated directly to the hardware structure that sets the program counter. Accordingly, the CPU calls the interrupt service routine without an intermediate step using an interrupt handler.

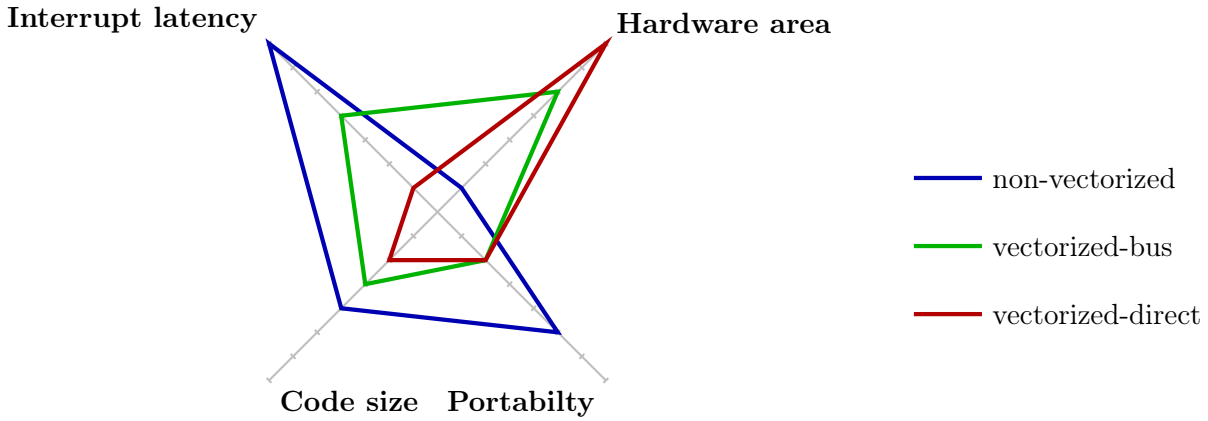


Figure 8.5: *Comparison between different interrupt modes on binary size, hardware area and interrupt latency.*

Figure 8.5 compares the different interrupt modes and points out where they are best deployed. The efficiency, measured by interrupt latency, refers to the time that elapses from the occurrence of an external interrupt until interrupt processing begins. Software primarily causes this latency during interrupt processing, e.g., semaphores, context switches, or the interrupt vector table processing. A non-vectorized PIC has performance penalties and requires more code size. This code size[45] increase is mainly because expensive bus accesses are required to resolve the interrupt vector table. However, the hardware footprint of the non-vectorized PIC is significantly smaller because the vectorized PIC requires additional registers and combinatorics to process the interrupt vector. A vectorized PIC has decreased portability since additional software wrappers are needed for specific design choices, e.g., generic driver architecture. For example, different interrupts may refer to the same generic ISR, as shown in Listings 8.1 and 8.2.

Listing 8.1: *Generic driver architecture with non-vectorized PIC.*

```
/************ Interrupt Handler ************/
void __irq__ mei_handler(void){
   if(bitfielde_get(IRQ1_ACTIVE){
      receive_uart(UART1);}
    else if (bitfielde_get(IRQ2_ACTIVE){
      receive_uart(UART2);}
}
/***************** UART *****************/
void receive_uart(UART_Config_t UART_Config);
```

Listing 8.2: *Generic driver architecture with vectorized PIC*

```
/***************** UART *****************/
void __irq__ receive_uart1_irq_wrapper(void){
   receive_uart(UART1);}

void __irq__ receive_uart2_irq_wrapper(void){
   receive_uart(UART2);}

/***************** UART *****************/
void receive_uart(UART_Config_t UART_Config);
```

[45] A larger code size also causes a higher memory requirement, i.e., it contributes to larger memory size.

Interrupt nesting A PIC that supports nesting temporarily disrupts a currently running exception by another one. So, the processor can faster serve higher priority exceptions. When a nesting interrupt occurs, the processor preempts the higher priority exception before finishing the current exception. Preemption is only possible if the interrupt is in a non-critical phase. A critical phase of a running interrupt is, e.g., during the context switches.

In RISC-V, the critical phase of the interrupt is protected by the global interrupt enable bit `mie` held in the `mstatus` CSR. The bit field controls whether an interrupt of a higher priority can be taken or not. The generated ISR routines set the `mie` bit after the context save phase to allow nesting of the ISR main task. Before entering the context load phase, it is cleared again. Special handling is required for critical ISRs, defined as non-preemptible, as they must be protected throughout execution regardless of their priority. Therefore, the generated critical ISR keeps the `mie` bit cleared for the entire process until the interrupt is returned. All interrupts that appear meanwhile wait until the interrupt is returned.

The performance of a PIC with nesting is purely application-specific and can be advantageous or disadvantageous. The main problem that nested interrupts may cause is that the CPU spends more and more time processing interrupts instead of doing other tasks. However, for many time-critical systems, nesting is essential to respond quickly to events. Another criterion that should not be underestimated is the complexity of the hardware. For example, a PIC that supports nesting requires about 250 more LUTs (3508 LUTs without nesting to 3766 LUTs with nesting) when defining four interrupt sources (increases with the number of interrupt sources).

8.2 Safety Pattern

Functional safety seeks to mitigate the level of risk in a function or the system. A functionally safe system identifies potentially hazardous situations and assures correct behavior despite systematic or randomized errors. As described in AUTOSAR [21], faults can be grouped into execution- and timing-related faults (e.g., deadlocks, incorrect execution sequence) caused by the processing unit. This framework introduces safety measures like watchdogs or PFMs (Program Flow Monitors) to verify the correct logical and temporal execution sequence.

Functional safety is commonly a neglected aspect of firmware generation frameworks. However, safety measures are best suited for model-driven generation systems as they usually define recurring structures. One of the reasons is that most safety measures require hardware support and are not software-only mechanisms, e.g., watchdog, redundancy checks, and assertions. Most firmware generation frameworks cannot deal with this. However, in the featured holistic HW/FW generation framework, the firmware generator is aware of all HW details and can thus create such HW-dependent safety patterns.

In order to include safety measures in the design, the generator backend deploys an endogenous safety transformation (also named safety pattern generator) on the abstract firmware model [221]. A designer can customize this safety transformation through safety patterns defined in the generator-spec introduced in Section 6.3. Figure 8.6 shows a snippet of the generator-spec used to instantiate safety patterns. On generation, the generator backend applies the safety transformation on the FW PIM as a function of the instantiated safety patterns. The result of this safety transformation is again an instance of the FW PIMM, but now containing the safety measure.

Note that this step of embedding safety patterns in the design preserves the previously-assembled functionality.

Furthermore, it is worth mentioning again that the safety transformation only extends the abstract firmware model in a device- and compiler-independent way. Accordingly, a safety pattern is not bound to implementation details and is reusable for different software layers, devices and compilers.

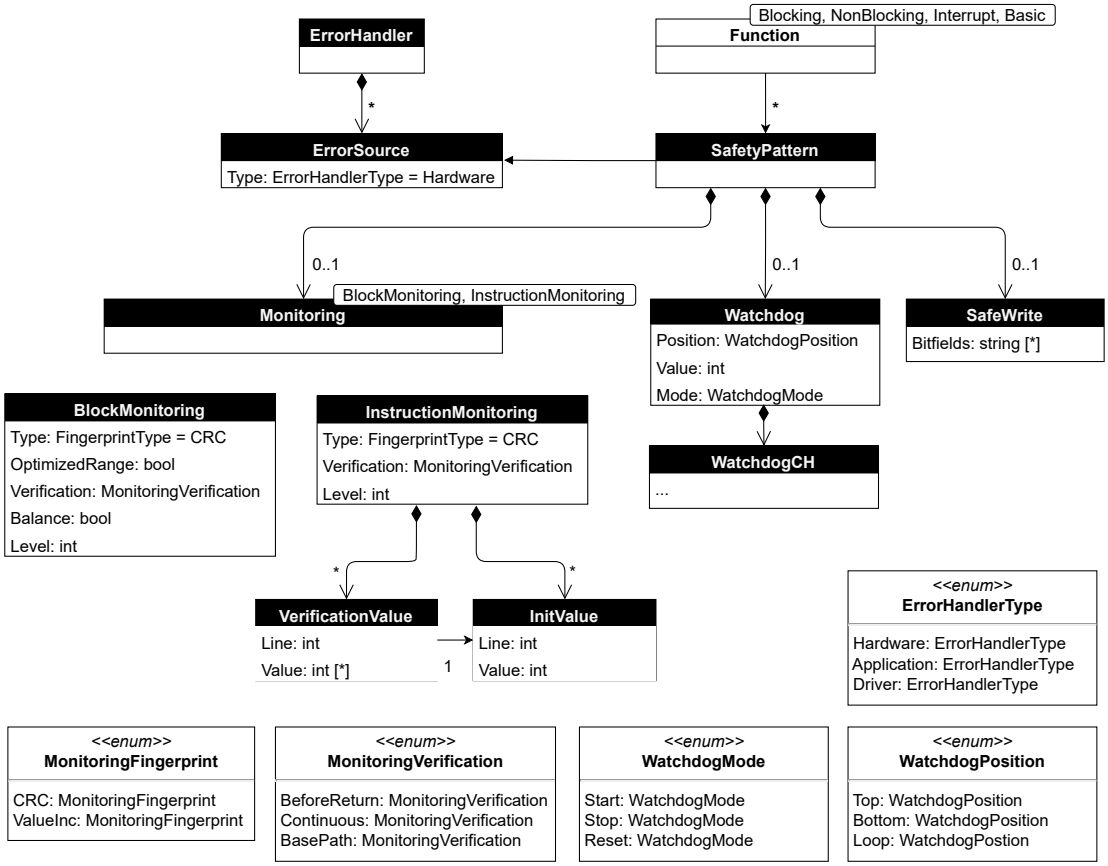


Figure 8.6: *Snippet of the safety relevant classes from the generator-spec metamodel of Figure 6.5.*

This section describes the concept of incorporating safety measures into an abstract model using three abstract safety patterns as examples: Watchdog, PFM, and redundant register checking. All those examples illustrate the framework's strength, which is its flexibility. Because of the pattern's configurability, designs with different safety measures can be generated. This diversity enables cost analysis (e.g., diagnostic coverage, hardware area and performance) and the adaptation of safety measures to the system requirements.

8.2.1 Redundant Register check after modification

A safety pattern protecting registers, suitable for introducing the principle of safety transformations, describes the redundant register check (in generator-spec: *SafeWrite*). This measure aims to control a write transfer to a register with subsequent read access and evaluation of the data. A potential error on the bus and the affected flip-flop inclusive access logic can be detected.

To deploy the safety measure, the designer specifies the list of *Bitfields* in the *SafeWrite* pattern that should be protected. During generation, the safety transformation is executed to apply the

redundant register check to the design. In the first step, the transformation traverses the entire behavior, all action nodes of the function, to find the particular write event to the bit field.

If such an event appears, it is replaced by a combination of action nodes, as shown in Listing 8.3. The safety pattern generator implements this safety measure only under the following conditions: The bit field must also be readable by software as well as the read access may not trigger a side effect.

Listing 8.3: *Cross check of register write accesses.*

```
uint8_t value_parity = 3;

setBitfield(PARITY_BF, value_parity);
uint8_t safe_write = getBitfield(PARITY_BF, value_parity);

if (safe_write != value_parity){
   error_handler(parity_check_error);
}
```

Following the idea and implementation of the redundant register check, other safety mechanisms can be added to the framework. A designer can implement various safety mechanisms through special safety pattern generators. A safety pattern generator wraps safety mechanisms around artifacts of the model. Such a generator approach is made in two steps: First, a detector identifies artifacts enveloped by safety measures, e.g., the protected register. Second, a safety transformation extends the abstract model with safety-relevant aspects, e.g., the write access is extended by a redundant check. In the following, this method is applied to further safety mechanisms.

8.2.2 Watchdog

The watchdog timer is a crucial component in safety-critical systems to protect the time-related program flow [22, 143]. A watchdog monitors the chronological behavior of the system and thus increases the system's resilience. The watchdog measure in software aims to prevent the watchdog timer from elapsing by resetting it at certain event-based checkpoints. A checkpoint may not be reached in a corrupted system (e.g., deadlock, blocking of executions), so the watchdog timer expires, triggering a reset or recovery mechanism.

A configurable *Watchdog* safety pattern is provided in the generator-spec. The system must contain a general-purpose timer[46] and an interrupt controller for implementing the watchdog. Both components can be customized and generated by the MDA framework. The *WatchdogCH* defines the channel of the general-purpose timer referenced by the watchdog that triggers an interrupt when it expires. The software controls the watchdog and resets its counting value at certain events. The system must ensure that the watchdog does not expire at any time.

A designer needs to define a timing sequence that maps all events that trigger a watchdog timer action. The generator-spec defines the time sequence as a list of functions, including a watchdog *Watchdog* event. Each watchdog event references a specific behavior triggered on the

[46]The RISC-V internal timer and instruction counter can also be used for the watchdog implementation instead of the general-purpose timer.

chosen watchdog channel. The configurable attributes of the watchdog are *Mode*, *Position* and *Value*.

The *Mode* specifies the type of action that controls the watchdog incorporated into the function. The list of modes to be selected includes: *Start*, which enables the watchdog and starts its execution; *Reset* feeds the watchdog with a new *Value*; *Stop* disables the watchdog. Furthermore, the designer can define the *Position* of the watchdog action in the function. The watchdog can be handled at function *Entry* or function *Return*. Additionally, the watchdog can be configured as *Wait* to be handled within the labeled control node. As shown in Section 8.1.2.1, this may result in the watchdog action being included in the busy-wait loop (blocking architecture) or before returning (non-blocking architecture). This setting makes the watchdog independent of the hardware performance, e.g., the baud rate of a communication device.

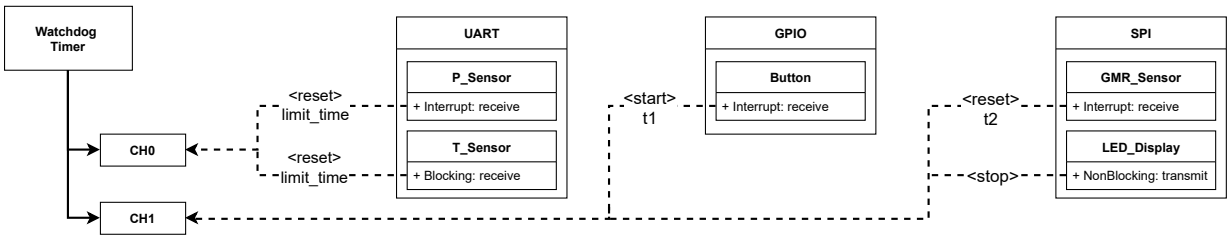


Figure 8.7: *Configuration of watchdog patterns in a generator-spec instance.*

Figure 8.7 illustrates an example with two watchdog timer channels (CH0 and CH1) while Listing 8.4 shows the generated code. With both channels, two separate time sequences can be defined for watchdog patterns. The first watchdog pattern referencing channel CH0 monitors whether data is received periodically via the UART interface from the pressure sensor *(P_Sensor* and temperature sensor *T_Sensor*. The UART's receive function resets the watchdog when data is successfully received and prevents the timer from elapsing under regular operation. In the example, the activation and termination of the watchdog are left to the application layer.

The CH1, meanwhile, is referenced by the watchdog pattern added to the GPIO and SPI functions. A button event triggers the initialization ($Value = t_1$) and the watchdog's activation. Within period t_1, the system must receive data from the GMR_sensor to reset the watchdog ($value = t_2$). Subsequently, the data must be transmitted to the LED display within t_2. Using the approach, the designer can specify any timing sequence by linking the functions of the firmware model and the timer channel.

Listing 8.4: *The generated code from the watchdog configuration of Figure 8.7.*

```
// UART.c
void receive_uart_P_Sensor_interrupt(){
    watchdog_reset(CH0); // Reset Watchdog
    ...
}
void receive_uart_T_Sensor_blocking(uint16_t* data){
    watchdog_reset(CH0); // Reset Watchdog
    ...
}
// PAD.c
void receive_GPIO_Button_interrupt(){
```

```
    watchdog_start(CH1, 20000); // Start Watchdog
    ...
}
// SPI.c
void receive_spi_GMR_Sensor_interrupt(){
    watchdog_reset(CH1, 20000); // Start Watchdog
    ...
}
void transmit_spi_LED_Display_nonblocking(uint16_t* data){
    watchdog_stop(CH1); // Stop Watchdog
    ...
}
```

8.2.3 Signature-based Program Flow Monitor

Execution-related errors such as control flow errors (CFE) are anomalies in the control flow, such as incorrect execution orders, skipped or suppressed instructions, or additional instructions. Figure 8.8a shows a control flow graph (CFG) with nodes v_n and edges e_n and three potential types of CFEs. Signature-based program flow monitors (PFMs) are common safety measures to alleviate such CFEs, as described in [13, 54].

A signature monitoring technique embeds signatures into the program during code generation or compile-time and verifies them during run-time. So, a PFM safety pattern extends the nodes of the control flow graph with two routines named "Signature Update" and "Signature Verification", as shown in Figure 8.8b. As the name implies, the "Signature Update" $s_{r+1} = U(s_r, u(v_n))$ calculates the next run-time signature s_{r+1} based on the preceding run-time signature s_r and a basic-block specific signature constant $u(v_n)$ assigned to the node v_n. The update routine specifies a signature function[47] such as CRC or MISRA. The "signature verification" $V(s, prior(v_n))$ verifies the correctness of the run-time signature. In this step, the run-time signature is compared with all compile-time pre-determined signatures of all potentially preceding nodes $prior(v_n)$. If no pre-determined signature matches the run-time signature, the PFM pattern reports a CFE.

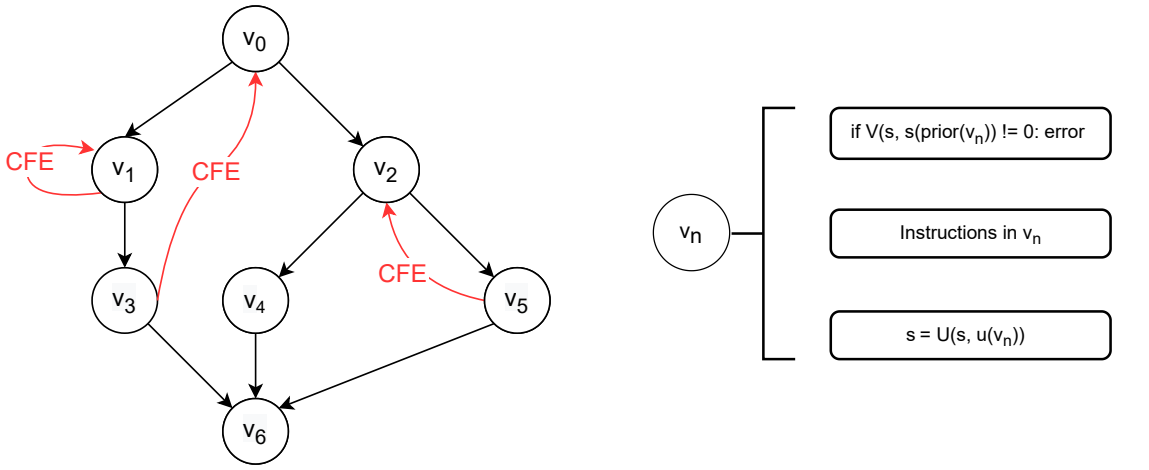
In general, a PFM monitors control transitions among the execution path to assure the correctness of the logical program flow and, thus, system resilience. However, PFMs can occur in various forms grouped into three granularity levels. The levels are defined by [13, 14] as inter-procedural PFM, intra-procedural PFM and instruction-stream PFM.

At the highest level of granularity is the inter-procedural PFM. That verifies the correctness of the calling relationships (represented as call graph) among functions. Such a measure aims to ensure that procedures are called and returned correctly according to the call graph. In this way, invalid function calls and returns can be detected. In order to build a call graph, detailed knowledge of the entire application behavior is required. This is beyond the scope of a framework that focuses on generating embedded software below the application layer.

The intra-procedural PFM, also called inter-block PFM, is a more fine-grained scheme that verifies the correct execution of basic building blocks[48] inside a function [227]. This measure is

[47] Werner et al. analyzed in [220] the functional requirements of signature functions and performed an evaluation on different signature functions for instruction-stream PFMs.

[48] A basic block describes a part of consecutive instructions with a single entry point as well as a single exit point



(a) *Control flow graph with potential CFEs.* (b) *Extension of a node by a PFM pattern.*

Figure 8.8: *Program Flow Monitoring concept in a control flow graph.*

a logical succession of the inter-procedural PFM, which examines the validity of execution using the function's CFG. It monitors the correct flow between building blocks by assigning signatures to each building block. This measure does not detect CFEs at all instruction levels, e.g., illegal jumps within the basic building block.

The instruction-stream PFM or intra-block PFM [174, 184] can deal with such errors and ensure the executed instructions' integrity. A building block contains an arbitrary number of sequential instructions. The instruction-stream PFM monitors that all instructions of the building block are properly sequenced at run-time. This granularity level requires that each instruction is assigned a signature. Compared to the two previous PFMs, the software does not trigger the "Signature Update" routine. Instead, the signature of each executed instruction is calculated in a dedicated module of the processor, as proposed in [220]. Finally, the calculated signatures can be verified, e.g., at the end of each basic block or function block.

There are several factors to consider when selecting the proper monitoring approach. In general, a pure software PFM such as [146] does not require additional hardware but causes a significant overhead in execution time to update the signature. Conversely, a hardware-assisted scheme such as the instruction-stream PFM requires special post-processing of the compiled binary [219, 220]. Furthermore, when choosing a PFM, it is important to note which error is a threat and needs to be handled.

8.2.3.1 Model Extensions

Figure 8.6 distinguishes between two PFM (*Monitoring*) concepts in the generator-spec that can be generated. The designer can apply either an intra-procedural PFM (*BlockMonitoring*) or a hardware-assisted instruction-stream PFM (*InstructionMonitoring*) to a function. In addition, depending on the safety requirements, the designer can further tailor the monitoring pattern, e.g., the monitoring depth or the hash function. These configuration options are discussed in the next sub-sections.

and only one execution path. All instructions within a basic block are always executed sequentially from start to finish, without branches or jumps.

In general, each function of the abstract firmware model is considered as CFG. To recap: In the FW PIMM of Figure 6.3, a function specifies an activity group as a control block, also named a basic block, which contains a list of actions. An action node can be a control node, which inherits another control block. Signature-based flow monitoring pattern generation is carried out in two phases. In the first phase, the pre-processing phase, the abstract function of the FW PIM is extended by a special flow monitoring artifact named *FlowMonitoring* node. The FW PIMM specifies a generic *FlowMonitoring* action node that can be inserted into any position within the activity. The node is treated as any other action node. It describes one of the following three manifestations:

- Signature *Initialization*: The initialization node pre-initializes the run-time signature with a specific signature constant. Upon this point, the signature can be updated along the execution path of the function. The initialization node is inserted at the beginning of the first basic block of the monitored activity.
- Signature *Update*: The update node calculates the next run-time signature based on the current run-time signature and a unique block signature constant.
- Signature *Verification*: The run-time signature is compared with the expected compile-time signature list.

The pattern generator inserts these action nodes into the control graph, ensuring coverage of all possible paths through the function. It considers each control block as a separate execution unit and places "Signature Updates" in each according to a defined set of rules.

In the second phase, the post-processing phase, the PFM generator determines all paths of the modified function's control flow graph that can be traversed from an initialization node to a verification node. In this way, a set of paths is obtained, with each path containing a different sequence of traversed update nodes. Together with the initialization node, the signature for each path can thus be calculated. Finally, the pre-computed signatures are appended as the compile-time signature list to the verification node.

8.2.3.2 Hardware Requirements

The PFM relies on a signature function such as the cyclic redundancy check (CRC) to calculate the run-time signature. In general, software computation of signature functions is feasible but involves significant performance costs. Therefore, a hardware PFM module shown in Figure 8.9 is added to the system, permanently active to compute the signatures. This HW unit is required to realize the PFM software pattern. The PFM is specified within the CPU generation framework and can be configured similarly to other IP components, e.g., by type of signature function.

The PFM module resides within the RISC-V core as the RISC-V pipeline controls it. The PFM module is accessed via the CPU's CSR interface. In particular, the instruction-stream PFM needs the pipeline status and the executing instruction as inputs to calculate the signature and to ensure the instruction stream's integrity. So, the PFM has to be connected to the execution stage of the processor's pipeline. In addition, the PFM registers, such as the real-time signature

register, are placed within the fast RISC-V CSR interface[49]. This location allows access to the registers by privileged atomic instructions and avoids slower multi-cycle bus accesses.

The intra-procedural PFM is integrated into the write paths of the following two CSRs, as shown in the architecture of Figure 8.9. First, the register *CSR_STATE* stores the run-time signature. This register is updated with the value of *csr_state_in* when initializing the signature via software command. Second, the virtual register *CSR_HASH* is used to compute the *signature_nxt*. While in intra-procedural PFM, the signature update event is triggered when writing to the *CSR_HASH* register via a software command. In the PFM instruction stream, this is triggered by an HW event. This event occurs for each instruction accepted and processed by the core. Both events cause the signature function to calculate the next *CSR_STATE* based on the *csr_hash_in* and the current value stored in *CSR_STATE*.

While checkpoints are calculated selectively in the intra-procedural PFM, they are calculated permanently in instruction-stream PFM. Accordingly, the PFM instruction stream architecture performs more hash calculations and therefore requires more power but can also handle more error cases. Moreover, the permanent calculation is faster since it is executed simultaneously with the processor's execution stage. However, the instruction-stream PFM needs more logic due to additional control signals.

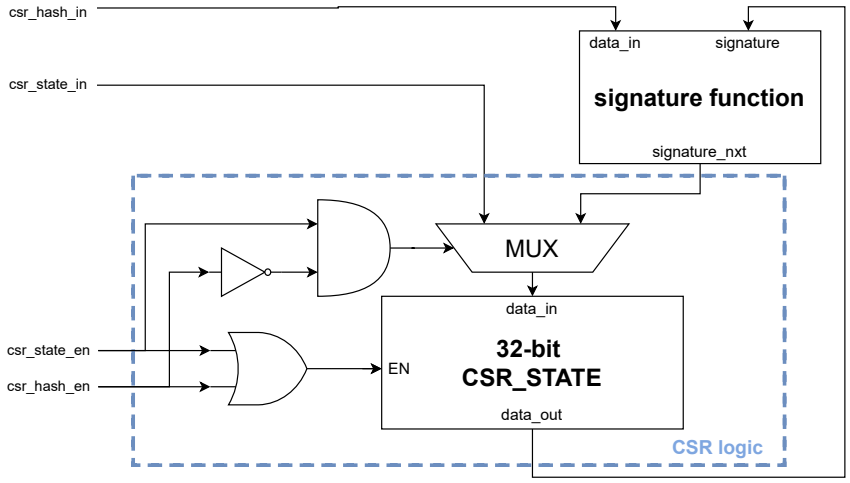


Figure 8.9: *The PFM hardware module included in the RISC-V core.*

8.2.3.3 Intra-procedural PFM

The safety transformation of an intra-procedural PFM pattern follows transformation rules that automatically extend the functional model with flow monitoring actions. The implementation details of this PFM are determined by the designer's configuration of the *BlockMonitoring* pattern in the generator-spec. With the MDA approach, a wide variety of intra-procedural PFMs can thus be generated. The intra-procedural PFM features the following options:

- **Type:** The *Type* defines the applied signature function.

[49]The custom CSRs for the PFM are specified in the user section of the CSR address range.

- **OptimizedRange:** The RISC-V CSR specification supports r-type (register operand) and i-type (immediate operand) instructions, as shown in Listing 8.5. An i-type write instruction (CSRRWI) uses a zero-extend unsigned 5-bit immediate. In contrast, the r-type write instruction (CSRRW) first loads the value to be written from the internal register, which requires additional instructions to be set. The designer can choose between both options, which affect the signature size used at each signature update node. Larger signature values are less efficient but provide higher coverage due to the larger bit-wise distance between values, reducing the likelihood of false negatives.

  Listing 8.5: *The generator applies the register or the immediate operand depending on the PFM configuration.*

```
// Swap value in the CSR Register (csr) and the internal register (src_r)
#define CSRRW(csr, src_r) __asm__ __volatile__("csrw %0, %1" :: "i" (csr), "r" (src_r));
// Write constant value (val) into CSR Register (csr)
#define CSRWI(csr, val) __asm__ __volatile__("csrwi %0, %1" :: "i" (csr), "i" (val));
```

- **Level:** Each function specifies a hierarchically nested control block. The *Level* specifies the depth of the nesting level that the PFM should protect. Adjusting the level can reduce the number of paths through all flow monitoring nodes. The signature path sp is defined as:

  Definition 8.8. $sp = \{\{v_0, ..., v_n\}, \{v_0, ..., v_n\}, ...\}$ *where* $\{v_0, ...v_n\}$ *is a sequence of flow monitoring nodes starting at the initialization node* v_0 *and ending at the final node* v_n*.*

  For example, Figure 8.10a shows a CFG with three nesting levels. Note that each control node introduces a further nesting level. Only the base path is monitored if the designer sets the level to 1; level 2 also includes the first nesting hierarchy. The whole list of paths for each level sp_{li} of the example is:

  $Level1 : sp_{l1} = \{\{v_0, v_1, v_6, v_7\}\};$
  $Level2 : sp_{l2} = \{\{v_0, v_1, v_2, v_6, v_7\}, \{v_0, v_1, v_5, v_6, v_7\}\};$
  $Level3 : sp_{l3} = \{\{v_0, v_1, v_2, v_3, v_6, v_7\}, \{v_0, v_1, v_2, v_4, v_6, v_7\}, \{v_0, v_1, v_5, v_6, v_7\}\};$

  As the level increases, the number of valid paths through the CFG grows. This increases the granularity of the monitoring and, thus, the diagnostic coverage. In return, the time spent by the CPU in the verification node to compare the run-time signature with each path's signature increases.

- **Balanced:** The *Balanced* setting aligns the signatures of different execution paths, improving the PFM pattern's performance. The objective of the approach is to match signatures to produce the same signature regardless of the execution path. Thus, only a single compile-time signature must be checked in the signature verification node.

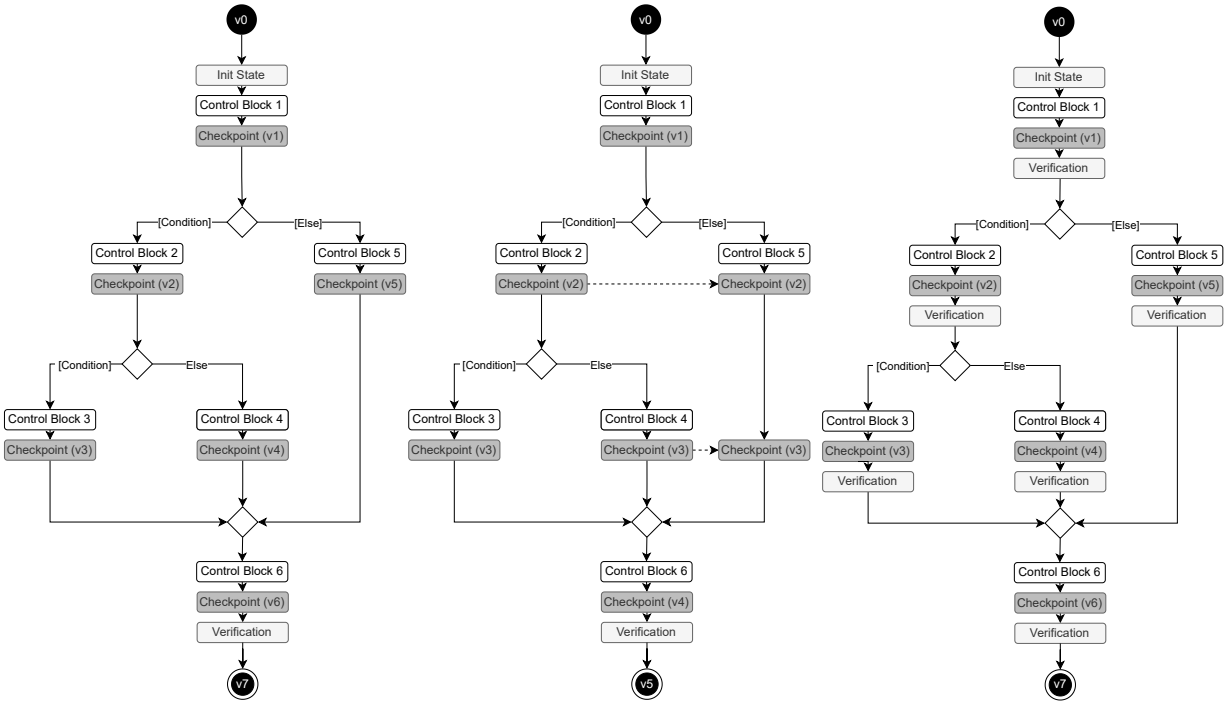
  As discussed in the preceding bullet, the CFG in Figure 8.10a yields three execution paths sp_{l3}, i.e., three valid signatures. Balancing the "signature updates" reduces the number of valid signatures to a single one without changing the execution paths, as shown in the CFG of Figure 8.10b. While the balancing approach improves verification effort, it also reduces diagnostic coverage since illegal jumps to parallel control blocks may not be detected.

- **Verification:** The PFM pattern supports three *Verification* options that affect the position of the verification nodes and thus the fault-tolerance latency (FTL[50]).

  – BeforeReturn: The generator builds a design, as shown in Figure 8.10a, which verifies the signature at a single point when leaving the function. This setting has a high FTL for complex functions.

  – Continuous: The signature is verified along the execution path after each signature update, as shown in the CFG of Figure 8.10c. *Continuous* verification minimizes the FTL.

  – BasePath: The BasePath is a trade-off between the previous two approaches. The verification nodes are distributed along the function's base path after each control node of its root group.



(a) *Default flow monitoring.* (b) *Balanced flow monitoring.* (c) *Low latency flow monitoring.*

Figure 8.10: *Different flow monitoring concepts are generated from the safety pattern configuration.*

Figure 8.11 shows the transformation steps for generating the intra-procedural PFM according to the designer's design choices. The transformation steps are explained in the following, taking the generated sample code of Listing 8.6 as a reference.

First, the generator must instantiate a random number generator (RNG) which populates the monitoring nodes. In particular, the "signature initialization" in line 3 and "signature update" in line 8, 12 and 15 utilize the RNG to write random constants to the registers. The RNG supplies integers according to the chosen maximum allowed integer size given by the *ReducedRange* setting.

[50]Fault Tolerance Latency (FTL) is defined as the total time taken from the occurrence of a fault to system's recovery. The FTL, according to [112], includes different fault-handling stages as error detection, fault location, system reconfiguration.

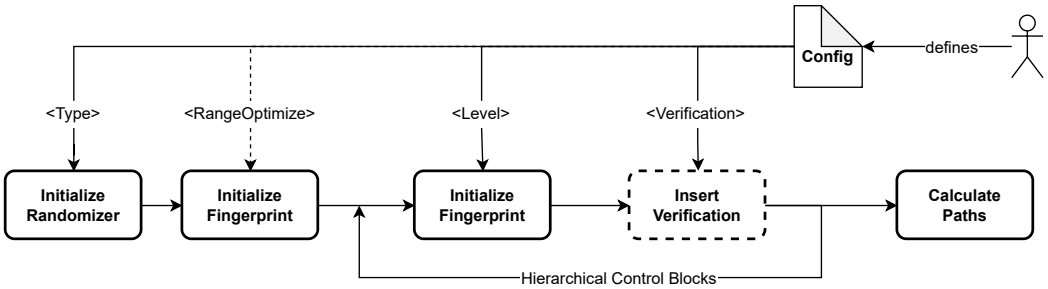


Figure 8.11: *Transformation steps to integrate a configurable intra-procedural PFM.*

The reduced range limits the random integer to 5 bits, while the long range allows all 32-bit integers.

The subsequent steps are performed in the first traversal phase, where the pure CFG is analyzed without monitoring nodes. In this phase, the generator adds monitoring nodes to the abstract model. The signature is initialized (`CSR_STATE_WRITE`) at the beginning of the function, while signature updates (`CSR_HASH_WRITE`) are included along the control path. The "signature update" is generally included in each control block before branching to a new control block, considering the configured *level*. The "signature verification" is optionally inserted at the end of each control block as defined by the *Verification* setting.

The signatures are balanced in an optional step that analyzes the CFG, including the monitoring nodes. The balancing step modifies the CFG to ensure that each execution path hits the same signature update nodes. This may imply that the signature constant for specific nodes needs to be adjusted or that additional "signature update" nodes must be included. In addition, minor changes to the graph may be required to satisfy the balancing requirement, e.g., a standalone if-statement must be extended by an else statement:

```
// Before balancing: Standalone If-Statement
if (condition){
    ...
    CSR_HASH_WRITE(325755)
}
...
```

```
// After balancing: Update in Else-Statement
if (condition){
    ...
    CSR_HASH_WRITE(325755)
} else {
    CSR_HASH_WRITE(325755)
}
...
```

The modified model is analyzed in the last step, and the compile-time signature list of all paths is calculated. This calculation is done in a second traversal step that iterates over the graph of the modified function to determine all signature paths *sp* between a signature initialization node and a signature verification node.

Together with the signature function, the compile-time signature list can be calculated[51] and included in the signature verification node, as shown in lines 17-26. This verification node processes the compile-time signatures and calls the error handler in case of a mismatch.

[51] Note that the signature function used by the generator to determine the compile-time signatures must follow the same concept as the one used in the hardware.

Listing 8.6: *Target code with a block-PFM pattern generated by the safety transformation template.*

```
1  void receive_uart(uint16_t len, uint16_t *data){
2      uint32_t _signature;
3      CSR_STATE_WRITE(2473823); //Signature is initialized with random value.
4      /* Behaviour of the function on the first control block level. */
5      CSR_HASH_WRITE(3463452); // Signature update before branch.
6      if (Condition){
7          /* Behaviour of the function on the sub-control block (if). */
8          CSR_HASH_WRITE(123482); // Signature update in if-branch.
9      }
10     else{
11         /* Behaviour of the function on the sub-control block (else). */
12          CSR_HASH_WRITE(456323); // Signature update in else-branch.
13     }
14     /* Behaviour of the function on the first control block level. */
15     CSR_HASH_WRITE(325755);
16
17     /* Verification of the correct control paths.
18     1. Path: 2473823->3463452->123482->325755 = 1910696930
19     2. Path: 2473823->3463452->456323->325755 = 3535675038 */
20     CSR_STATE_READ(_signature);
21     static const uint32_t _valid_signatures[] = {1910696930, 3535675038}; // Calculated paths
22     static const uint8_t _valid_signatures_len = 2;
23     bool _valid = verify_signature(_signature, _valid_signatures_len, _valid_signatures);
24     if (False == _valid){
25         /* Error Handler */
26     }
27 }
```

A special handling in the transformation steps is required for non-deterministic loops in the CFG, as shown in Listing 8.7. The control block within the non-deterministic loop is considered a standalone CFG that is transformed independently of the previous and subsequent program flow. To this end, the same transformation steps are applied again to the loop only. Furthermore, the run-time signature is buffered when the CPU executes the loop.

Listing 8.7: *Transformation of a non-deterministic loop for an intra-procedural PFM.*

```
CSR_STATE_READ(_signature); // Store signature
while (condition){
    ... // PFM pattern applied on the CFG of the non-deterministic loop.
}
CSR_WRITE_Write(_signature); // Load signature
```

8.2.3.4 Instruction-stream PFM

Instruction-stream PFM differs from intra-procedural PFM because it is a hardware-assisted methodology. The "signature update" routine is not triggered by a software event (writing to CSR_HASH_WRITE) but runs concurrently with the program execution. Instead of a random constant value, the HW PFM module uses the instruction encoding of each executed instruction from the CPU pipeline to calculate the next signature. However, the initialization of the signatures and the verification remains further under the control of the software.

Indeed, the compile-time signatures of each execution path can no longer be derived from the abstract firmware model. The execution paths and, thus, the compile-time signatures are obtained from the disassembly file of the compiled target code. Therefore additional steps in the PFM pattern generator are required.

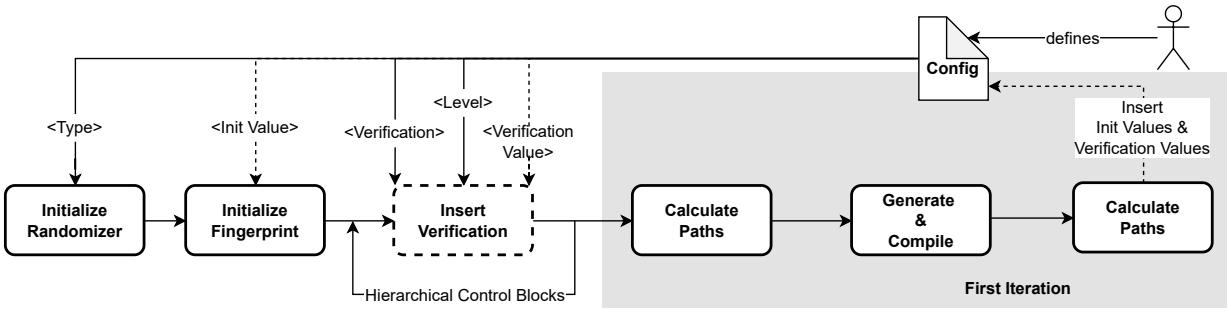


Figure 8.12: *Transformation steps to integrate a configurable instruction-stream PFM.*

The transformation flow of the instruction-stream PFM is shown in Figure 8.12. Note that the transformation steps specified in the flow are performed in two iterations. In general, each run is similar to the intra-procedural transformation, except that no signature update nodes are inserted into the model, and thus the balancing step is skipped. Also, the state of the signature is buffered for any non-deterministic event in the control flow graph.

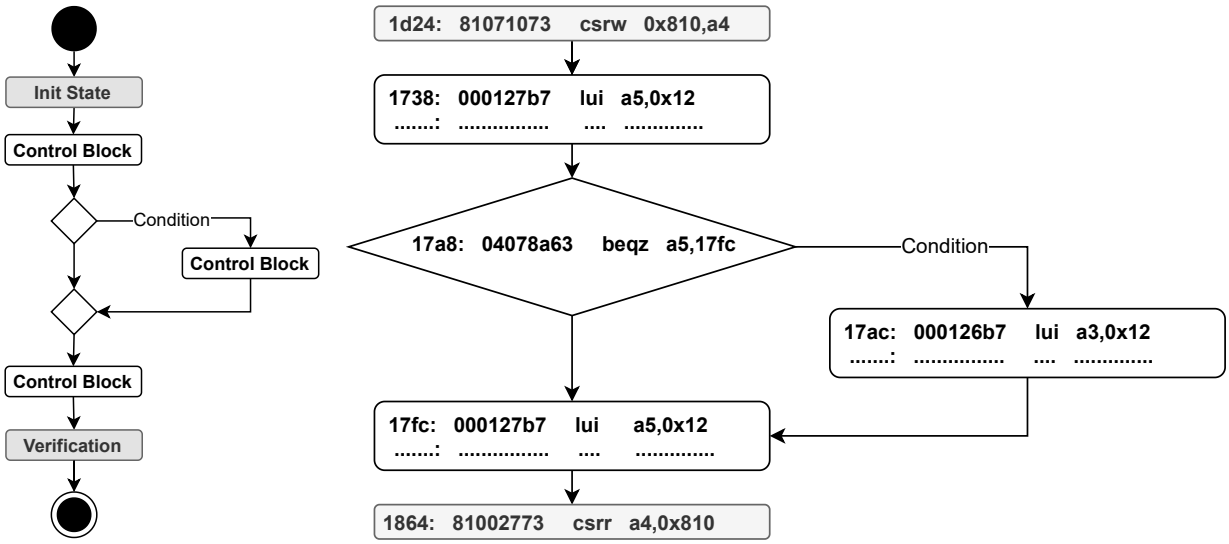
In the first iteration, the steps are performed according to the chosen configuration following the previous flow of the intra-procedural PFM. When calculating the number of paths from initialization to verification node, the generator considers the abstract model, as shown in Figure 8.13a. The number of paths indicates the number of compile-time signatures to be checked at the verification node. Those compile-time signatures are initially not subject to any calculation and are populated as dummy values.

The target code has to be generated and compiled with predefined compiler settings to derive the correct verification values. The control paths with all included instructions are reconstructed from the disassembly file.

The extraction is done by a parser that determines the control flows based on the instruction encoding, as described in [14, 204, 212]. The parser extracts all paths starting with the "CSR_STATE_WRITE(...)" instruction that initializes the PFM and ends with the verification node: "CSR_STATE_READ(...)". The enclosed instructions are dropped if the state is buffered due to a non-deterministic event. In this way, all possible sequences of instructions from initialization to verification are obtained, as shown in Figure 8.13b.

From these sequences, the compile-time signatures can be calculated. Each signature is calculated based on its initialization value and the encoding of all instructions in the sequence using the signature function of the PFM module. The calculated signatures and the initialization values are stored in a temporal file.

In the second iteration of the safety transformation, the initialization and verification nodes are filled with the values stored in the temporal file and not with random values. Thus, the same transformations are executed in both iterations resulting in the same target code but different compile-time signatures. As a final step, the instruction sequences and the correctness of the previously determined paths must be verified after another generation and compilation step. If



(a) *Abstract CFG* (b) *CFG reconstructed from the disassembly file.*

Figure 8.13: *Compilation flow of the instruction-PFM pattern.*

the paths differ from the determined ones of the first iteration, the generation has failed, and the designer will be notified.

A crucial factor is that the firmware model and the compiler configuration are identical for both iterations. In addition, the CFG of the abstract function and the assembler must match to ensure that the correct number of valid paths can be determined. Otherwise, an error occurs, e.g., due to aggressive compiler optimization.

8.2.4 Error Handler

Each generated safety pattern refers to an error-handling routine invoked upon an error occurrence. The generator-spec defines the *ErrorHandler* as a group of different *ErrorSources*. An *ErrorSource* specifies the appropriate course of action upon an error detected by the safety pattern, as demonstrated in line 25 in Listing 8.6. In this context, multiple safety patterns can thus refer to one error source, accordingly one error response.

Listing 8.8: *An enumeration of all error sources associated with the system's safety patterns.*

```
typedef enum error_source_t{
    ERROR_HANDLER_NONE,
    ERROR_HANDLER_SPI_rx,
    ERROR_HANDLER_UART_rx,
    ERROR_HANDLER_I2S_tx,
    ERROR_HANDLER_I2C_tx,
}error_source_t;
```

The generation of the error handler is defined in a separate transformation template. This generator constructs an enumeration in Listing 8.8 containing all possible error sources. In addition, the generator builds skeletons that manage those error sources, leaving it up to the designer to implement the best method to deal with the error manually. The following *ErrorHandlerTypes* define the implementation detail of the error handler:

- *Hardware*: The hardware layer handles the error and triggers a system reset in case of an error source. This reset can be caused by an illegal instruction or an architecture-specific reset such as the "EBREAK"[52], which is inlined into the safety pattern's error response:

```
inline static void error_handler_hardware(error_source_t err_src){
    __asm__ volatile ("EBREAK"::)
}
```

- *Application*: When an error occurs, a global variable is written by the safety pattern to indicate the source of the error `error_source_t error_source`. Instead of executing a direct error handling routine, this routine is moved to the application layer. On this layer, the `error_source` can be queried permanently to ensure the system's integrity. However, this increases the FTL.

```
error_source_t error_source;
inline static void error_handler_application(error_source_t err_src){
    error_source = err_src;
}
```

- *Driver:* This handler is also intended for the application developer to decide on the error response. Unlike error-handling in the application layer, this approach reduces FTL since the error handler is called immediately after the error occurs. The generator builds the error handler as an if-else skeleton with all error sources, which is called by the error routine of the safety pattern.

```
void error_handler_driver(error_source_t err_src){
    if ((err_src == ERROR_HANDLER_watchdog_uart_sensors))
        // Fill manual code here
    else if ((err_src == ERROR_HANDLER_watchdog_button_display)
        // Fill manual code here
}
```

The designer can individually choose each function's safety pattern approach and thus provide multiple error-handling methodologies in the system. Note that the designer must consider the FTL when taking a particular approach.

8.2.5 Results

Safety-critical designs should be rigorously verified to ensure functional correctness and safety as outlined in the ISO-26262 standard [98]. This standard recommends fault injection as an essential technique for the safety verification of safety-critical designs. As defined by [114], fault injection can be defined as the deliberate insertion of faults into the system. There are various fault injection techniques such as hardware-based, simulation-based and emulation-based. The proper system's response to an injected error can prove the reliability of the safety patterns.

[52] In general, the "EBREAK" instruction in RISC-V was primarily designed to cause an execution stop by the debugger. However, the "EBREAK" is also used to mark code paths that should not be executed leading to an hard reset.

Safety patterns increase diagnostic coverage but also influence cost requirements. The following detailed analysis measures the cost impact of each safety pattern, including performance, memory footprint, and diagnostic coverage.

8.2.5.1 Fault Injection

Proving the reliability of safety patterns requires introducing a fault injection mechanism. Simulation-based techniques are a standard method for demonstrating diagnostic coverage. Kaja et al. [103, 104] present a fault simulator that extends Verilator[53] [186], an open-source hardware simulator with fault injection capability. This extension of Verilator simplifies fault injection and captures the system's response.

A fault simulator [196, 200] generally manipulates the values of internal signals during simulation time by applying fault models. Accordingly, the system can be put into a state as if a hardware error has occurred. After that, the system can be observed to determine the impact of the introduced error on the safety pattern report. In order to verify the resilience of the system and its safety patterns, thousands of randomized failure models have to be applied. Verilator's strength lies in the fact that it is a cycle-based high-performance simulator.

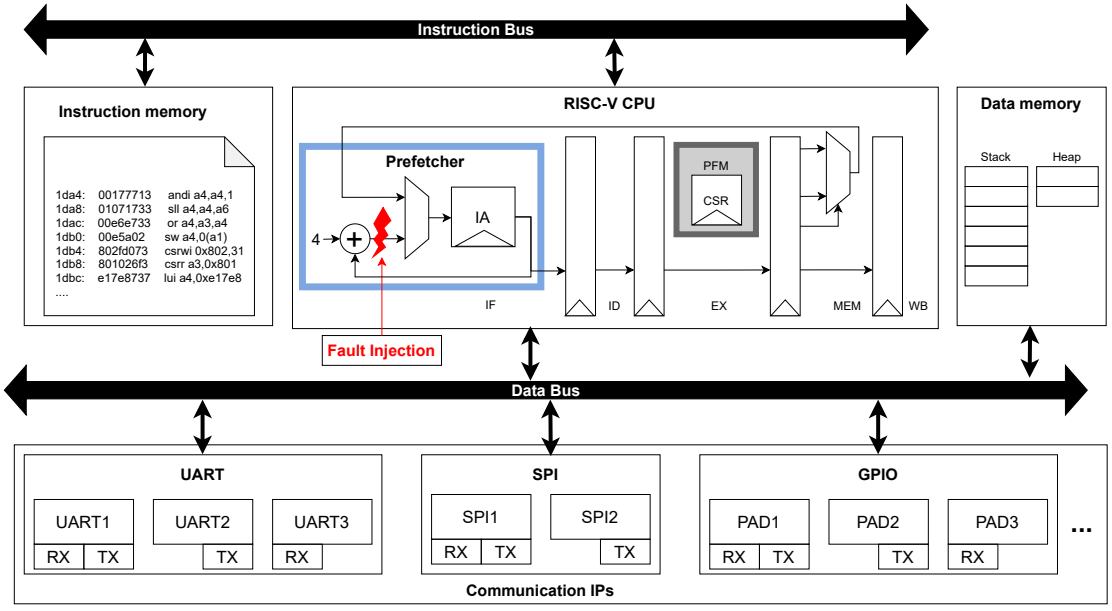


Figure 8.14: *A SoC including the CPU pipeline stage with the fault injection concept.*

In order to verify each safety pattern, various errors in hardware are examined that manipulate the program counter (PC). Therefore, different fault models are injected into the design, such as permanent (stuck-at) and transient faults (bit flips). Correct behavior increments the PC value by four if no control flow instruction is present. A fault model weaved into the PC register, as shown in Figure 8.14, leads to an incorrect execution order in simulation and, thus, faulty behavior.

For example, to validate the PFM and watchdog, a fault model is weaved into the CPU pipeline to manipulate the logical behavior and temporal execution order. More precisely, the

[53] Verilator employs a cycle-accurate behavioral model that computes the logical state once per clock cycle. Compared to event-based simulators, Verilator is super-fast but cannot observe intra-period glitches.

hardware logic responsible for calculating the PC is manipulated. The framework, as described in [104], allows the injection and definition of different fault models, such as permanent (stuck-at) and transient faults (bit flips). Figure 8.14 shows the SoC with the fault injection concept in the pipeline stage. With this concept, the error is inserted into the feedback loop of the address calculation. As a result, the next retrieved instruction address (IA) changes, and the execution order deviates from the expected order. If no fault is fed in, the PC is incremented unless a jump instruction is present.

For example, suppose the PC has the value 0x000C, and a bit flip is injected in the next clock cycle (CC). In normal behavior, the PC takes the value 0x0010 in the next clock cycle; error injection takes an erroneous value instead, e.g., 0x0110. A PFM or a watchdog is intended to detect such an error.

8.2.5.2 Diagnostic coverage and overhead

The fault simulation is performed at the RTL level of the SoCs described in the next chapter 9. The applications executed on the system, including the safety patterns, were compiled with RISC-V GCC 9.2.0 using optimization level -01. For a good indication of diagnostic coverage, an analysis was performed with 3000 fault models. Each fault model is simulated, and the result of each simulation is stored in a dump file. This data is further analyzed to determine the proportion of detected faults and the FTL.

First of all, the implementation of the watchdog is evaluated. A watchdog implementation requires a general-purpose timer IP in hardware and a small overhead in device drivers. The binary size increases, and performance decreases due to the instructions required to initialize, reset, and stop the timer. Compared to the PFM, the watchdog has limited configurability. The watchdog's reliability and diagnostic coverage have been measured using stuck-at-fault models on the PC register. A stuck-at-fault is critical when it results in an execution loop that updates a watchdog in each iteration. However, employing multiple watchdog channels can avoid this kind of fault scenario. The watchdog implementation achieved an average diagnostic coverage of 95.7% for a single watchdog channel, while five watchdog channels achieved more than 99.9% coverage. The FTL between watchdog expiration to execution of the error handler depends on the interrupt controller design introduced in Section 8.1.2.2.

8.2.5.2.1 Transformation capabilities The strength of this approach, similar to the driver design, is that a safety pattern is configurable and can result in various variants. The designer can customize 18 variants of instruction-stream PFM and even up to 72 intra-procedural PFMs. The effort to realize all variants is limited to the one-time implementation of the safety transformation (337 SLOCs for intra-procedural PFM and 631 SLOCs for instruction-stream PFM). The generator frontend remains untouched. Thus the developer can focus exclusively on implementing the functionality of a new IP component without worrying about safety details.

Therefore the generator approach saves much development time compared to manual coding. More importantly, it is more reliable since the manual implementation of PFMs is a very repetitive and thus error-prone task. Table 8.3 shows the average number of SLOCs generated for different IP devices. The overhead for manual implementation of the intra-procedural PFM is the largest,

as it takes care of initializing, updating and verifying the signature by FW. Instruction-stream PFM neglects the signature updates but requires more SLOC to buffer the signature for non-deterministic events. The difference between the IPs' overhead arises from the number of IP options that lead to more execution paths.

Table 8.3: *Evaluation of the effort required to code safety patterns for different IP instances.*

Peripheral	Generated device driver (⌀ SLOC)			Device-specific Transformation (SLOC)
	Block-PFM	Instruction-PFM	Without	
UART	929	869	760	302
DMX	1055	993	859	377
I2S	667	649	549	489
SPI	1026	956	820	471
PAD	630	430	548	440

8.2.5.2.2 Memory and performance overhead Table 8.4 shows the achieved result for each configuration of the block-PFM using the CRC as a hash function. The results describe the average result of the safety pattern applied to all driver functions of various SoCs.

Each PFM configuration incurs different overhead in terms of binary size (ranging from 20,0% to 48,2%) and performance (ranging from 12,2% to 31,1%). The largest contributor to overhead is the verification node, which iteratively compares the run-time signature with all compile-time signatures. The evaluation reveals that configurations resulting in more compile-time signatures or adding more verification nodes have significantly more overhead.

For this reason, the overhead between instruction-stream PFM and intra-procedural PFM differs slightly since both verify the signature via firmware commands. However, there are different hardware costs: A CPU without the PFM concept requires 3904 lookup tables (LUTs). With intra-procedural PFM, the number of LUTs increases by 5,3% and with instruction-stream PFM, even by 8,6%.

8.2.5.2.3 Diagnostic coverage According to [133], between 33 % and 77 % of the errors, a computer system can suffer lead to a control flow deviation. The PFM addresses these control flow deviations. Table 8.4 shows the diagnostic coverage for bit-flips, stuck-at-0 (1000 CC) and stuck-at-1 (1000 CC) faults. The described fault injection approach can randomize[54] faults by time as well.

Similar to the overhead, the diagnostic coverage also strongly depends on the configuration. For example, the configuration determines the proportion of the monitored code. The diagnostic coverage is thus 14% higher if all the drivers' nesting levels (*level=0*) are monitored. The analysis further shows that the diagnostic coverage of a bit-flip ranges from 57,2% (62,0% monitored code) to 81,5% (89,2% monitored code). Whereas the FTL depends on the number of verification nodes. The results show that the FTL for the *Continuous* configuration is, on average 54 CC (37.6%) smaller for bit flip faults. Also, balancing reduces the FTL enormously (40%) but at the expense of diagnostic coverage (-3%).

[54]Note that the randomized error can occur throughout the run-time, even in execution points that are not monitored.

Table 8.4: *Diagnostic coverage and overhead analysis for different intra-procedural PFM concepts: R=BeforeReturn, B=BasePath, C=Continuous; Level=0 covers all nesting levels, Level=1 covers only first nesting level*

Configuration				Bit flip		Stuck-at-0		Stuck-at-1			Overhead	
Balance	Optimized	Verification	Level	Diagnostic coverage	FTL average (CC)	Diagnostic coverage	FTL average (CC)	Diagnostic coverage	FTL average (CC)	Monitored code	Binary size	Performance
True	Optimized	R	0	69.5%	209	65.6%	892	69.7%	502	75.0%	24.5%	15.6%
			1	57.2%	122	54.2%	783	59.3%	404	62.0%	20.0%	12.2%
		C	0	81.5%	106	77.9%	803	82.3%	233	89.2%	45.7%	30.3%
			1	62.5%	131	62.3%	848	64.4%	263	70.1%	32.2%	25.4%
		B	0	77.0%	179	74.4%	849	78.7%	378	81.8%	34.7%	24.7%
			1	58.9%	158	57.6%	825	58.5%	286	67.5%	28.3%	19.4%
	Extended	R	0	71.8%	247	72.5%	947	69.5%	475	76.6%	26.9%	16.5%
			1	59.9%	144	60.3%	830	60.1%	392	62.9%	21.4%	13.8%
		C	0	85.5%	120	86.1%	825	87.0%	234	90.8%	48.2%	31.1%
			1	67.0%	139	68.6%	830	63.7%	301	71.0%	33.5%	27.4%
		B	0	78.6%	160	77.7%	847	77.8%	305	83.7%	37.5%	26.6%
			1	63.9%	166	60.1%	858	63.8%	467	68.5%	29.7%	21.3%
False	Optimized	R	0	68.0%	136	7.7%	831	65.2%	356	73.4%	22.0%	14.2%
			1	57.3%	122	53.3%	799	54.2%	302	62.0%	20.0%	12.2%
		C	0	79.1%	34	79.4%	720	79.3%	184	85.7%	40.5%	26.7%
			1	66.3%	68	62.7%	759	62.9%	203	70.1%	32.2%	25.4%
		B	0	73.3%	90	73.5%	772	69.6%	298	79.1%	30.7%	21.9%
			1	64.5%	98	61.3%	773	61.7%	236	67.5%	28.3%	19.4%
	Extended	R	0	71.1%	130	69.9%	888	69.9%	352	75.9%	25.9%	16.2%
			1	59.4%	113	56.1%	801	59.4%	384	62.7%	21.0%	13.7%
		C	0	81.2%	40	78.9%	727	78.7%	238	88.1%	44.1%	28.7%
			1	67.3%	79	65.3%	766	65.6%	283	70.8%	33.2%	27.0%
		B	0	78.4%	107	75.2%	780	76.6%	308	82.2%	35.2%	25.4%
			1	63.8%	117	64.4%	800	60.4%	343	68.3%	29.5%	21.6%

The diagnostic coverage for stuck-at-1 errors is slightly larger than bit-flips because the error is preserved over the observed time windows. Diagnostic coverage of stuck-at-0 errors is worse because they can lead to an execution loop that gets stuck in a control block. Such a loop may neither reach a verification node nor a signature update node.

The main disadvantage of intra-procedural PFM is that faults causing jumps within the scope of a control block cannot be detected. Instruction-stream PFMs can cope with these faults and achieve an average of 8% more diagnostic coverage.

8.3 Optimization

One of the great strengths of code generation and meta-modeling is that many SoC variants can be generated. The framework presented so far includes several design choices the designer can

make. Briefly, a designer can configure the top-level requirements of the SoC and its IPs. He can also configure the architecture and design-specific preferences as well as safety patterns for the particular IP instance. Additionally, he can customize the register interface and explore different memory layouts, as shown in Chapter 5. Finally, the designer can select the code generator that translates the abstract models into the desired target language with a specific implementation style.

All these configuration options within a generator-based model-driven framework enable the creation of millions of variants. This diversity leads to an enormous amount of simulation data, which also opens up the possibility for design exploration and trade-off analysis. The insights gained from these explorations further enable the application of design exploration. Referring to [36, 92, 148], machine learning in EDA can become a state-of-the-art approach to design optimization. However, the major obstacle of machine learning in EDA is the lack of training sets [72]. For this reason, algorithms are primarily used that are particularly suitable for small training sets [92]. However, the demonstrated MDA framework provides the training sets and thus enables the application of machine learning and algorithms suitable for larger data sets.

In the following, two applications of design optimization, compiler optimization and memory layout optimization, are described.

8.3.1 Compiler Optimization

So far, the compiler is an overlooked aspect of generation frameworks, but with a lot of optimization potential. A compiler translates the target code into a binary file loaded into the SoC's memories. Elaborated compilers, such as the GCC [78], provide hundreds of optimization flags for tuning the translation step.

In general, each compiler flag potentially causes a trade-off that affects costs like performance, power and memory overhead. It is challenging to define the best combination of compiler flags. On the one hand, compiler flags are interdependent and influence their effectiveness. On the other hand, the efficiency of optimization flags varies depending on the application and CPU (e.g., instruction set). Indeed, a designer can thus choose optimization flags that are not at all optimal for the respective applications.

Furthermore, certain compiler requirements must be met, especially when optimizing embedded software that interacts strongly with the hardware. For example, hardware accesses are often tied to hardware events and may have timing constraints. Rearranging the accesses through optimization could cause a malfunction or an invalid race condition. In other words, too aggressive optimizations can cause unexpected errors affecting hardware and software interaction.

Compilers like GCC generally provide predefined optimization levels, e.g., -O1, -O2 or -O3, which activate a set of optimization flags [101]. With an increasing level, the optimization gets increasingly aggressive, up to the point of breaking the standard's compliance (-Ofast). Although they are designed for various design spaces, these optimization levels achieve, on average, a good result [79]. For this reason, there remains further potential for application- and hardware-based fine-tuning, as described in [50, 53, 135, 156, 208].

In [223], a compiler flag optimization step is introduced, which is incorporated into the generation pipeline. The idea of the compiler optimization step is to adjust the default optimization

level so that it is optimal for a given design space. For this purpose, the step identifies all flags of the optimization level that have a negative impact on costs. It uses multi-criteria exploration through clustering methods to identify those relevant compiler optimization flags.

This optimization step helps to tailor the compiler to the generation framework, hardware, and firmware. Furthermore, it helps to quantify the potential memory and performance gains at an early stage. For example, the information obtained from such a step can be used to adjust CPU or memory specifications, e.g., through a feedback loop.

8.3.1.1 Compiler integration and training set generation

A designer can specify 2^{104} different optimization flag combinations for the RISC-V GCC v9.2.0 compiler. A common approach to overcome the huge parameter space is iterative optimization, as presented in [53]. The iteration order in such an optimization approach can be determined by a random number generator [37] or machine learning (ML) [18, 76, 145]. The ML-based technique reduces the exploration time but cannot fully compete with the results obtained by iterative compilation.

Similar to the approaches of [79, 156], the compiler exploration step is done in reverse order starting from an aggressive optimization level. The -O3 level is considered the starting level, which describes the superset of the -O1 and -O2, including the largest number of enabled optimization flags. The optimization step successively determines the flags that reduce costs when disabled, keeping -O3 active. This has the great advantage that dependencies between flags must not be resolved. For example, optimization flag *-fschedule-insns2* only takes effect if *-fschedule-insns* is also enabled. If the flags are elaborated starting from no optimization level, *-fschedule-insns2* is incorrectly evaluated as ineffective. In the reverse iteration, however, the impact of the flag *-fschedule-insns2* on the cost becomes apparent when disabled *-fno-schedule-insns2*.

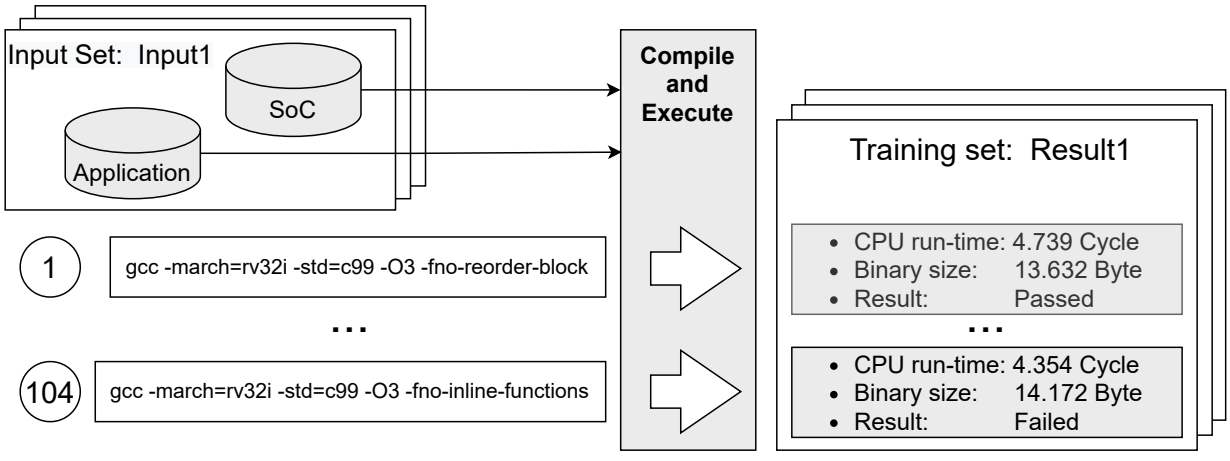


Figure 8.15: *Exploration of compiler flags as part of the generation framework.*

In order to explore the cost impact of all flags for a generated RISC-V SoC and a specific application, the framework automatically builds the binary for each disabled optimization flag. Therefore a compiler command is constructed and executed for each flag to measure its impact, as shown in Figure 8.15. Then, the SoC including the binary is simulated to determine the following objectives:

- CPU run-time: The application's execution time in clock cycles.
- Binary size: The binary size of the compiled application.
- Trade-off: A 50%-50% trade-off after normalizing the results of the two metrics, binary size and CPU run-time.

Two things are crucial for the application to determine the CPU run-time. First, the application must specify a start and end event that defines the measured time frame. Second, the application must verify that the execution was successful. Section 9.3 shows the implementation of these two conditions.

8.3.1.2 Clustering-based compiler exploration in nonlinear optimization space.

This approach has been extended to determine the best compiler flags as a function of design decisions described in the previous sections, e.g., device driver design, HAL or safety measures. Investigating the effect of compiler flags on design decisions requires a set of data sets (training sets). So, the previously described compiler flag exploration is performed on different applications and SoCs.

The framework uses clustering as an unsupervised learning task to interpret the data sets and determine suitable compiler flags. Clustering is a machine learning approach that looks for hidden structures in the data to group them by characteristics [90]. For compiler flag optimization, this is well suited because no effort is required to categorize the data[55]. Thus, the method can easily adapt to new applications, compilers, design decisions or compiler flags without much effort in categorizing them. Clustering analysis can determine the relevance of certain flags to CPU run-time, binary size, or both.

The k-means clustering [129] suits the compiler optimization problem as it is straightforward to implement, provides high scalability for large data sets, and can be easily adapted to new samples. Elkan's algorithm in [69] was preferred over the naive implementation to speed up the k-means clustering.

One drawback of k-means remains, which is the number of clusters that have to be considered by the algorithm. The exact number of clusters is, in fact, a hyper-parameter and must be specified in advance. As a solution to this issue, the well-known Elbow method [205] is applied, which helps to find the right number of clusters.

In summary, the Elkan k-means clustering method combined with the elbow method enables identifying the flags of interest for a particular CPU and a set of design decisions. These flags, if disabled, have a high impact on the optimization (in terms of performance and memory footprint). A detailed discussion of the approach using the elbow method and Elkan's k-means algorithm is provided in [223].

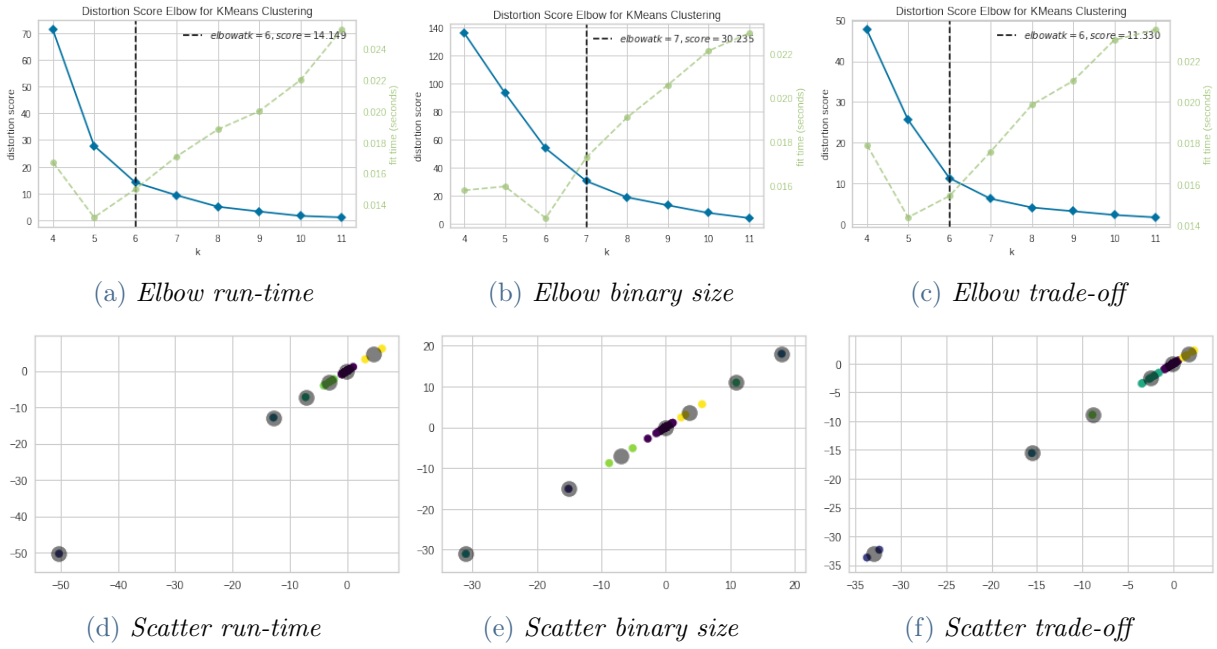
8.3.1.3 Evaluation

This section gives an overview of the compiler flag evaluation applied to a particular processor and set of design decisions. In particular, the presented analysis is done for a five-stage RISC-V

[55] Note that cost of memory demand and execution time is extracted automatically.

processor supporting RV32I base ISA. The generated firmware contains no safety patterns and follows a generic driver implementation with synchronous behavior. The compiler version used in this evaluation is the GCC v9.2.0. Python 3.7 with scikit-learn 0.23.1 and yellowbrick 1.1 was used to implement the considered methods. The methods were executed on the Intel Core i7-8700K CPU and a DIMM 32GB DDR4-3000 module of RAM.

The analysis is done in two steps for each objective (CPU run-time, binary size and both). First, the clustering algorithm is fed with n applications (input sets), yielding in $n \times 104$ compilation and execution runs. The k-means clustering is applied to this training data with the number of clusters obtained from the elbow method. Second, the test applications (test sets) are applied to evaluate the trained model.



(a) *Elbow run-time* (b) *Elbow binary size* (c) *Elbow trade-off*

(d) *Scatter run-time* (e) *Scatter binary size* (f) *Scatter trade-off*

Figure 8.16: *Evaluation of elbow method and k-means clustering for different cost requirements.*

The outcome of the elbow method is shown in Figures 8.16a–8.16c. The dashed vertical line indicates the number of clusters K. ($K_{time} = 6$, $K_{size} = 7$ and $K_{both} = 6$). Next, the clustering is performed with the determined K on the training set, as shown in Figures 8.16d–8.16f. The percentage of optimization for different clusters is highlighted (in grey, the centroids).

Table 8.5 lists the results of the test set when the identified flags are disabled (*Disabled Flags)*. The average optimization achieved takes as a reference the achieved optimization in level -O3. The results for each optimization objective are discussed below:

- CPU run-time: The -O3 optimization is already geared towards performance optimization. Nevertheless, the performance can be slightly improved by deactivating the suggested flags. On average, the run-time can be improved by 3.1%.

- Binary size: The potential for improving the binary size is significantly larger with -O3 as the reference optimization level. Thus, an average reduction of the firmware size of 20.93% is achieved.

	CPU run-time	Binary size	Trade-off
Disabled Flags	*fcrossjumping* *fguess-branch-probability*	*ftree-loop-optimize* *finline_small_functions* *finline-functions*	*fcrossjumping* *ftree-vrp* *ftree-fre* *ftree-slp-vectorize* *fguess-branch-probability*
Average Optimization	3.10%	20.93%	1.75%

Table 8.5: *Achieved optimization compared to basic level -O3 through disabling the identified compiler flags applied to the test sets.*

- Trade-off: The majority of flags affect both costs. Consequently, the optimization gains are lower (1.75%) than the single-object optimization.

The evaluation results show that disabling flags can greatly improve the default optimization levels of GCC.

8.3.2 Memory layout optimization

Chapter 5 discusses the generation flow for the register interface. Briefly, this chapter highlights the flexibility of the register interface metamodel, which allows the configuration of various memory layouts (mapping bit fields in registers). Most generator approaches let the designer configure all possible options to specify the IP fully. This configurability is highly important as it impacts the performance and the area of the IP containing the RI.

In [190], the memory layout was purely optimized to speed up the performance of register accesses. The register interface can be tuned to address several cost factors in a holistic generation flow. This memory layout optimization step is hidden from the user, similar to compiler optimization. The framework uses ML-based techniques inspired by computer vision detailed in [179, 180, 181, 230] to predict the cost of memory layouts. Such a prediction obviates the need for time-consuming synthesis in the exploration phase, speeding up cost exploration by a factor of 600. Besides prediction, an automatic optimization step based on Deep Reinforcement Learning deals with the multi-objective optimization problem to identify a better RI configuration [178, 182, 183].

8.3.2.1 Training set generation

In order to build the training set, a generation and analysis pipeline is created. This pipeline randomizes instances of the register interface metamodel from Figure 5.1. So, each instance has a different register layout. Apart from the layout, each instance also defines different access sequences.

In the training phase, the actual costs of each instance are determined. The framework runs the RTL synthesis of Vivado[56] as a batch process to obtain the synthesis area report. This report

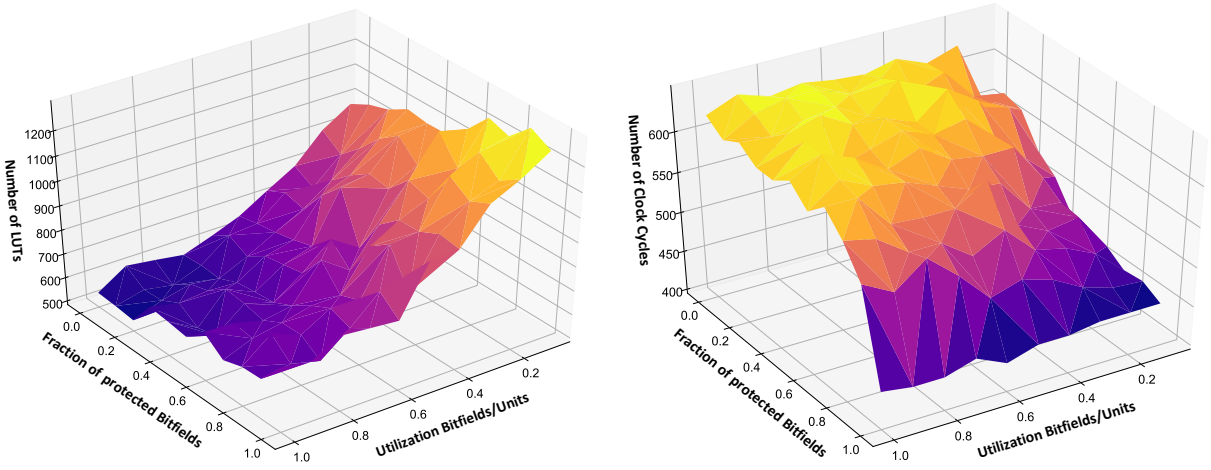
[56]The used version of Vivadio is the Design Suite 2018.2 and the synthesis is done for the Arty-7 FPGA board from Xilinx. The documentation on Vivado and generated reports can be found on the Vivado Design Suite User

provides good insight into the actual area requirement in terms of lookup tables (LUTs) and flip-flops. The actual binary size of the HAL, including all memory macros and register accesses, is retrieved from the disassembly file[57]. The actual run-time result is obtained from the simulation.

The Figures in 8.17 show the results of the costs for different memory layouts for the Safety Demonstrator - Human-Robot Interaction application presented in Chapter 9. The diagram illustrates the total footprint and run-time of all RIs used in the demonstrator. The performance and area of the RI depend strongly on the utilization of the registers and the number of protected BFs (in the RI metamodel: *ParityBit*).

Figure 8.17a demonstrates that a compact BF allocation results in a much smaller design since less control logic is required. In turn, protected BFs lead to a larger area because it requires extra logic to evaluate the validity of the write access. However, the performance analysis in Figure 8.17b shows an inverse proportionality to the logic area. A compact BF mapping leads to an increased number of costly RMW accesses. Conversely, an increasing number of protected BFs positively affects the performance.

A designer is confronted with a huge design space due to a large number of memory layouts. The designer can choose configurations for the given application, yielding a range from 527 up to 1213 LUTs. Similarly, the run-time ranges from 411 up to 623 clock cycles. This potential highly motivates the need for automatic memory-layout optimization.



(a) *Area analysis* (b) *Performance analysis*

Figure 8.17: *Evaluation of costs for different memory layouts.*

8.3.2.2 Register Interface estimation and optimization

The idea of cost prediction is to give an accurate estimation without performing the required compiler/synthesis steps. The approach uses ML-based techniques from computer vision (CV) to estimate the cost. In general, an image is extracted and interpreted in CV by associating specific features with a particular context, e.g., pattern recognition.

The task of cost estimation for the RI's memory layout can be mapped to the CV domain

Guide 2018.2.

[57]The disassembly file was created using RISC-V GCC 9.2.0 with optimization level -O1.

[180, 181]. In this domain, the RI characteristics are associated with the costs. Accordingly, the same algorithms can be applied. Instead of an image containing the information of the individual pixels, a RI-specific data structure is used. This data structure comprises the properties of the RI metamodel, e.g., bit field properties, mapping information or access sequences.

After the CV model is trained with the generated training set, the association between costs and RI features is present in the trained model. As evaluated in [230], the trained model can speed up the prediction time by 600 times. Moreover, the hardware area prediction has more than 98% accuracy, while the firmware size and CPU run-time prediction reach 85% [180, 181].

On top of the prediction flow, the framework is extended by an optimization step. This step, described in [178, 182], uses deep reinforcement learning (DRL) to analyze the data structures and predict the best RI configurations. In [183], different DRL algorithms are compared. The optimization results obtained with the best approach led to an average improvement of 12.2% in area, 11.0% in firmware size, and 10.7% in run-time.

Chapter 9

Application

Modeling and generation is always a profitable approach if it can automate repetitive tasks. This is the case with the generation of low-level embedded software, as explained in detail in the previous chapters. The application layer generally does not satisfy this requirement since the space of applications is enormous and cannot be mapped as an abstract specification model. For example, a system with a single LED already offers a large range of application variants, e.g., different types of light sequences.

Nevertheless, it may also be reasonable to describe a generator for a concrete application, especially when the application appears in several manifestations. The application generator takes the same approach as a driver generator since the FW-PIMM and the language generator, e.g., for C, are not restricted to a particular embedded layer. The application generator's purpose is once again to map an instance of the application as an FW-PIM, which is further translated into the target language. The following three features are added to the generator framework to facilitate modeling and analysis of applications:

- The FW-PIMM is extended by a finite-state machine, widely used as a design pattern for application behavior.
- External libraries must be supported unless the entire embedded software is derived from the framework. In other words, externally declared types, functions and other objects must be able to be referenced through the application generator. To this end, a parser is employed to extract these declarations.
- The framework is extended by verification as well as performance measuring nodes. These nodes are intended for the analysis of the application.

This chapter introduces three examples to prove the applicability of the approach. One of these examples is a self-test application in different manifestations, verifying generated designs and two industrial demonstrators that successfully apply the model-based generation framework to more complex use cases.

9.1 State Machine

The application of a reactive (event-driven) embedded system interacting with its environment using sensors and actuators is often organized as a finite state machine (FSM). An FSM describes a logical process as a sequence of events and actions that depends on the system's environment (event) and the state machine's history (state).

The basic concept, as described in UML 2.0 in [164], defines a process through a list of states, whereby the process is always in exactly one of these states. A state change is caused by a well-defined conditional trigger that causes the transition. The FW-PIMM follows the notation of the UML FSM in Figure 6.3, which defines the FSM through a set of states and transitions. The actions associated with the states and transitions are modeled as a sequence of actions described in Activities and Actions. The FSM generally distinguishes three types of action allocations[58]:

1. The action associated with the transition.
2. The *Do*-action is executed as long as the process remains in the state.
3. The *Else*-action is executed when no transition takes place.

The generator follows two different state-centric implementations of the FSM[59]. First an FSM that follows a nested if-else structure. Second, an FSM based on function pointers. Both options offer the same functionality, although the implementation is different, resulting in varying benefits.

As a rule, the generator chooses the implementation variant of the FSM based on the number of states. An if-else FSM is preferred for smaller FSMs with less than five states since a function pointer implementation is design overkill. On the other hand, the function pointer FSM implementation is more suitable for more states. The designer can also bypass this general rule and specify his preferred implementation option when configuring the generator

9.1.1 Nested if-else FSM

An FSM breaks down complex problems into manageable states and state transitions. A switch or if-else construct is the most common and safest implementation of an FSM. Listing 9.1 shows the generic implementation of an if-else-based FSM created by the generator. The generic implementation details the three positions where actions are included in the FSM. The *Do*-actions in Line 8 are triggered while the FSM is in the state. A state change triggers specific transition-related actions, e.g., Line 10 and 14. Whereas the *Else*-actions in Line 19 are executed in the absence of a transition.

This kind of implementation may lead to spaghetti code with an increasing number of states since the logic is kept in one block. Also, performance decreases with larger FSMs due to the increasing number of jump instructions. In contrast, the advantages of an if-else FSM are that it occupies less static memory and is safer as it does not involve polymorphic behavior. Besides, it is easier to debug.

[58] The UML specifies two types of state actions executed upon state entry and state exit. Both types are not essential for a typical FSM design and are thus not considered in the FW-PIMM.

[59] Note that there are innumerable other ways to implement FSMs, such as a table-based FSM.

Listing 9.1: *A generic generator result of a FSM based on if-else.*

```
typedef enum {S_<label1>, S_<label2>, ...}state_t;

int main(){
    state_t state = <entry_state>; // Initialize state variable with entry state
    while(state != terminate_state){
        // look for event
        if (state == S_<label1>){
            // Do action
            if (event1){
                // Transition actions for S_<label1> to S_<label2>
                state = S_<label2>;
            }
            else if (event2){
                // Transition actions for S_<label1> to S_<label3>
                state = S_<label3>;
            }
            // Further events for S_<label1>
            else{
                // Else actions
            }
        }
        // Further states
    }
}
```

9.1.2 Function pointer based FSM

The function pointer FSM, as shown in Figure 9.2, breaks the entire FSM into state functions. The state function array `FSM[]` defines the mapping between the states and their respective functions. The run-time state is a global variable that assigns one of the possible values of the `state_t` enumeration. Note that the enumeration has to be in sync with the state function array.

The FSM calls the function associated with the current state in the main loop. A state function focuses on the actual state-related actions and reduces unnecessary jumps required for the high-level flow of the FSM. Compared to the if-else-based FSM, unnecessary control jumps can thus be omitted. In summary, the performance increases with such a design, but at the cost of additional static memory.

Listing 9.2: *A generic generator result of a function pointer based FSM.*

```
typedef enum {S_<label1>, S_<label2>, ...}state_t;

struct fsm_type_t{
    state_t state;
    void (*func)(void);
};
// Maps a state to its state function
struct fsm_type_t FSM[]={
 {S_<label1>, &S_<label1>_func},
 {S_<label2>, &S_<label2>_func},
```

```
11   ...
12 };
13 // Global state variable
14 state_t state;
15
16 void S1_<label1>_func(void){
17     // Do action
18     if (event1){
19         // Transition actions for S_<label1> to S_<label2>
20         state = S_<label2>;
21     }
22     else if (event2){
23         // Transition actions for S_<label1> to S_<label3>
24         state = S_<label3>;
25     }
26     // Further events for S_<label1>
27     else{
28         // Else actions
29     }
30 }
31
32 void main(void){
33     state = <entry_state>; // Initialize state variable with entry state
34     while(state != terminate_state){
35         // look for event
36         FSM[state].func(); // Call the function associated with the current state
37     }
38  }
```

9.2 External Software

A generator approach is valuable when the design is not static but is reused in another variant. This does not necessarily apply to the application layer. A pure generator approach at this level can be overkill since external libraries or very application-specific software libraries are often employed. Furthermore, establishing a new framework in existing structures poses the challenge of maintaining support for proven concepts. For embedded system design, manually implemented IPs still need to be integrated as part of a system. Therefore, the traditional way of IP design must work seamlessly with the generator approach.

For this reason the framework contains a parser, which extracts the interface of the "external" or manually written C or C++ file. The parser integrated into the process is the open license pycparser[60] [31]. The global declarations of the functions, variables and types are extracted from these files and cached as a dictionary. Thus, the generator is aware of all available software elements and can cross-reference them in the transformation scripts.

Once a manually coded function is referenced by the CIM to PIM transformation script, the presence of the function is checked during generation. A warning is thrown whenever the function

[60] Pycparser is a python module that parses C code into an AST which can serve as a front-end for, e.g., analysis tools.

is unavailable, or the call parameters do not match. This way, the developer can rectify the application before compiling the source.

9.3 Analysis of application

High-level IP models connected with an automatic generation scheme for hardware and firmware enables design space exploration. Section 8.3.1, presented two approaches for design space exploration: Compiler Optimization and Register Interface Optimization. In this context, two requirements are crucial. First, invalid designs must be properly identified, and only verified designs should be considered in the exploration process. Second, fast and sufficiently accurate metrics have to be used to quantify performance uniformly.

To accomplish both, the FW PIMM establishes wrapper classes used to label the parts of the application that are under test. Both wrappers are applied during the test phase or design space exploration of the SoC. A verification wrapper specifies rules that must be followed when executing the labeled activity. Similarly, a performance wrapper is applied to measure the execution time of activities. The idea of both wrappers is to store their results in a reserved RAM address range for testing during execution. This address range is written to a report file at the end of the simulation, ready for further investigation.

9.3.1 Firmware Verification

An automated approach to variant verification is important to ensure the correctness of each possible design decision, e.g., at the specification level. Beyond that, an automatic verification approach also enhances the stability of such a framework. Software RTL simulation on the system-level is a holistic approach to verify both firmware and hardware to a high confidence level. The FW-PIMM establishes a verification class as a wrapper to verify the activities of interest. The developer allocates a set of activity- and specification-dependent rules to this class, which are applied during the SoC's test phase or design space exploration.

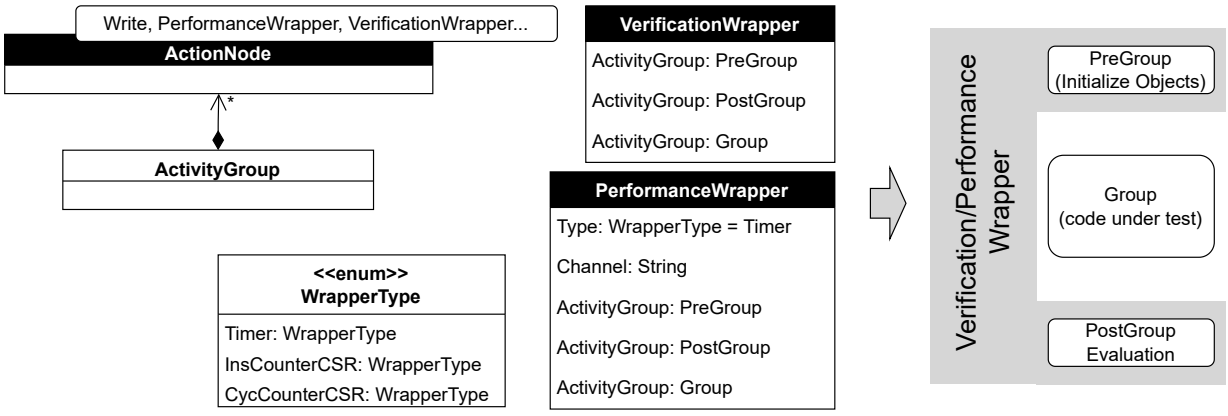


Figure 9.1: *Integration of the wrappers into the FW-PIMM, snippet from 6.3.*

A *VerificationWrapper* in Figure 9.1 specifies an action node that comprises three activity groups. Since the wrapper is an action, it can be inserted into an existing activity just like any other action node. The three subgroups of the wrapper are defined as follows: Pre-group that

carries out actions to initialize the verification wrapper, followed by the group that specifies the activity to be verified. The post-group defines the measures comparing the activity's outcome against the golden model's expectations. The pre-group and the post-group are excluded from the generation when running the generation flow for the product.

Listing 9.3 shows the verification wrapper's implementation on the Montgomery multiplication example. It demonstrates the previously described structure with the three groups that must be implemented.

Listing 9.3: *Implementation of a verification wrapper in a transformation script.*

```
# Verification Wrapper
pre_group, main_group, post_group = group.addVerifWrapper(Label="MontgomeryMul")

# Pre-group - Initialize Variable
b_int = 7854451
e_int = 4572177
m_int = 5641457

b = pre_group.addCreate(In=sVariable(Name="b", Type="uint32_t", Value=b_int));
e = pre_group.addCreate(In=sVariable(Name="e", Type="uint32_t", Value=e_int));
m = pre_group.addCreate(In=sVariable(Name="m", Type="uint32_t", Value=m_int));
c = pre_group.addCreate(In=sVariable(Name="c", Type="uint32_t"));

# Main-group - Montgomery multiplication: c = b^e mod m
montgomery = self.getActivity("montgomery_mul")
montgomery.setIns(b,e,m)
main_group.addCall(In=montgomery, Out=c)

# Post-group - Result
main_group.addVariableVerification(Result = c, Expected = pow(b_int, e_int) % m_int)
```

The resulting code from the example is given in Listing 9.4. The pre-group, as well as the main-group, is resolved in the usual way. Instead, a marker (`INSERT_MARKER`) is introduced in the post-group. It stores the value of interest at a specific address location inside the test address space. Note that the code does not directly execute the comparison between the expected and resulting values. The comparison takes place after the end of the simulation.

Listing 9.4: *The generated C code in test mode for the example Listing 9.3.*

```
//*************** START OF VERIFICATION WRAPPER *********************;
//*************** pre-group *********************;
uint32_t b = 7854451;
uint32_t e = 45721770;
uint32_t m = 5641457;
//*************** main-group *********************;
uint32_t c = montgomery_mul(b, e, m);
//*************** post-group *********************;
INSERT_MARKER(539648, c);
//*************** END OF VERIFICATION WRAPPER *********************;
```

9.3.2 Performance Analysis

A similar approach is followed for performance measurement as for verification. However, compared to the verification approach, the generator back-end automatically populates the pre-group and post-group. So, the designer only needs to provide the activity of interest and the kind of performance measurement when designing the generator. The *PerformanceWrapper* in Figure 9.1 defines distinct *Types* of performance measurement.

One option is to utilize a separate *Channel* of the general-purpose timer for the measurement. A second option uses the machine timers and counters available in the M-mode of the RISC-V [16]. The *mcycle* CSR records the number of cycles the processor has executed, while the *minstret* CSR captures the number of successfully executed instructions. Both counters are incremented automatically while the processor is running[61]. Furthermore, both counters have 64 bits to keep track of the counter's value. So, each machine counter is depicted by two 32-bit registers, with the upper 32 bits stored in the *mcycleh* respectively *minstreth* CSR. An example of a performance wrapper is outlined in Listing 9.5.

Listing 9.5: *The generated C code of a performance wrapper utilizing the machine counter.*

```
1  //*************** START OF PERFORMANCE WRAPPER *********************;
2  //*************** pre-group ***********************;
3  uint32_t t0_lower = CSR_READ(mcycle);
4  uint32_t t0_upper = CSR_READ(mcycleh);
5  //*************** main-group ***********************;
6  uint32_t c = fibonacci_result = fibonacci(521);
7  //*************** post-group ***********************;
8  uint32_t t1_lower = CSR_READ(mcycle);
9  uint32_t t1_upper = CSR_READ(mcycleh);
10 INSERT_MARKER(539652, t0_lower);
11 INSERT_MARKER(539656, t0_upper);
12 INSERT_MARKER(539660, t1_lower);
13 INSERT_MARKER(539664, t1_upper);
14 //*************** END OF PERFORMANCE WRAPPER ***********************;
```

9.3.3 End-to-End analysis flow

The basic concept of the wrappers is that their result is written into a particular address range during the simulation. A simulation process, controlled and executed by a batch script (or test bench), can analyze this address ranges at the end of the simulation. Both the generation of the batch script and the evaluation of the results can be automated. Thus, the analysis process can be directly interfaced with the MDA generation flow, as shown in Figure 9.2.

An essential part of the analysis flow is the creation of a *wrapper.txt* file next to the target code. This file holds the reference addresses of the used wrappers to store their result, e.g., time-stamp or values of the monitored variables. This information is needed to create the batch script that starts the simulation and evaluates the memory once the simulation is finished. The evaluation result is then provided as a report (.rpt) file.

[61] The counters are only active if the enable bit is set in the *mcountinhibit* CSR. When disabled, e.g., the user has no need to read the counter, the machine counter is paused to save power.

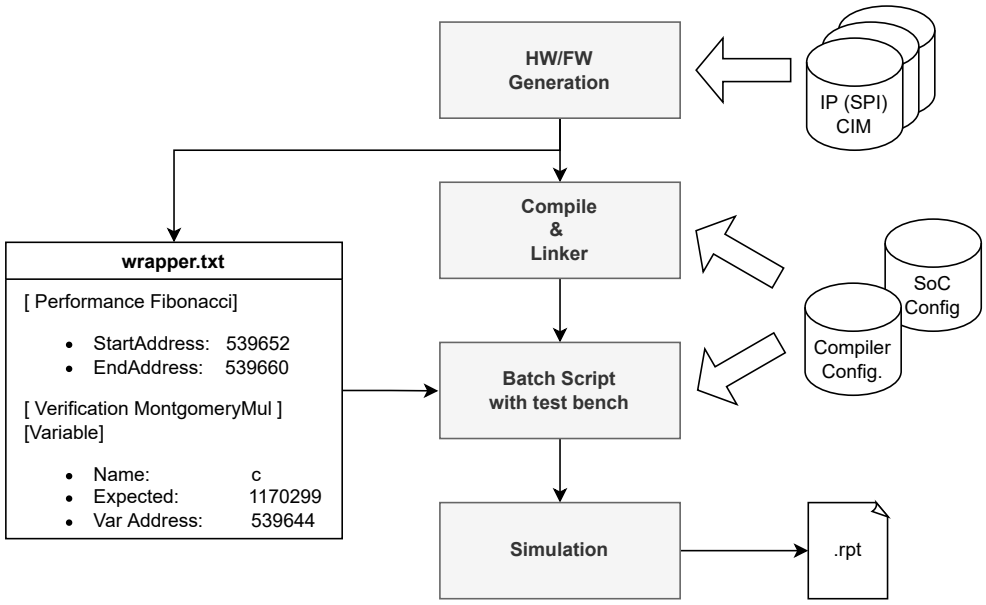


Figure 9.2: *End-to-end flow from generation to performance evaluation and verification.*

The framework supports two simulators, namely Vivado and Verilator. In Vivado, the simulator is addressed by a .tcl script, while Verilator requires a `C++` script for controlling the simulation. In principle, Verilator has the advantages of a higher simulation speed [163] along with the ability of error simulation [104, 199].

In summary, this automatic flow provides excellent opportunities for design exploration. For example, a designer can compare the impact of different RISC-V extensions, compiler settings (in Figure 9.3) or hardware design decisions. Also, the integrity of safety measures for different compiler settings can be checked, as well as many other use cases.

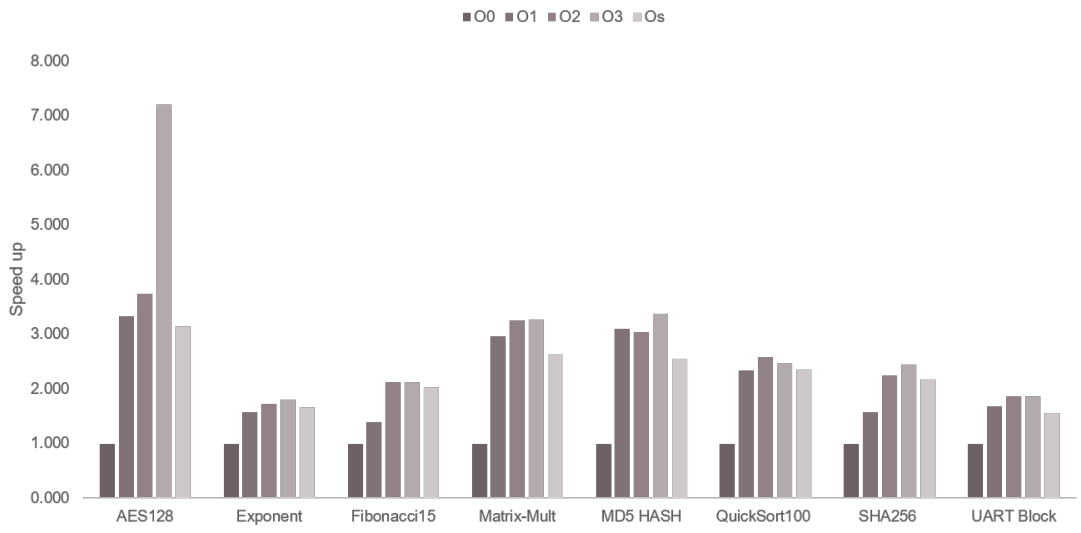


Figure 9.3: *Run-time comparison between different compiler levels.*

9.4 Proof of Concept

The framework's approach is verified by practical applications running on an FPGA (Arty Z7-10). The hardware for these applications, synthesized on the FPGA, is built using the embedded systems automation approach [68]. The presented embedded software generation flow incorporates into this MDA flow and closes the design gap of missing software support for the generated SoCs.

The following introduces two demonstrators and a Self-Test SoC, proving the generation framework's quality. Both demonstrators show that the methodology is also suitable for more complex industrial applications. One application describes an ML-based mechanism for location detection. The second application is from the domain of human-robot interaction. The Self-Test Soc, instead, is mainly used to verify the generation framework together with different IP and embedded software variants.

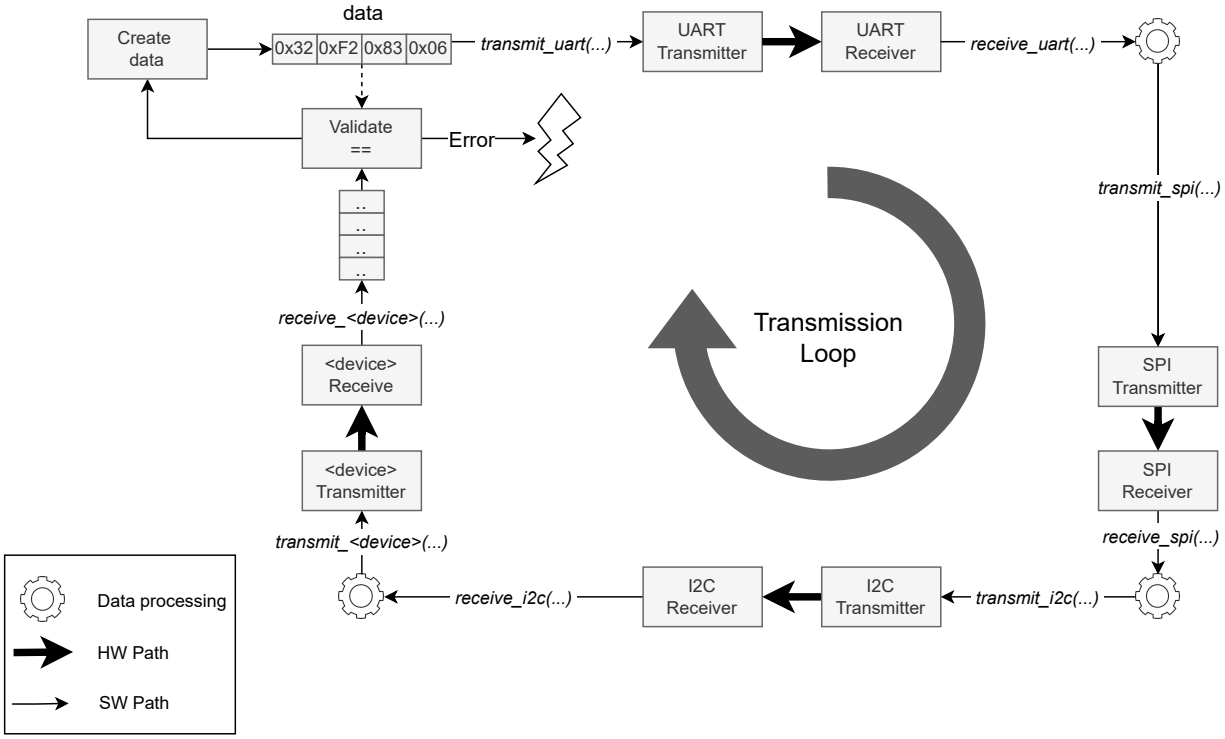


Figure 9.4: *Self-Test SoC with receiver-transmitter pairs.*

9.4.1 Self Test System-On-Chip

A Self-Test SoC is applied to verify the different configurations of the IPs. The overall idea of the Self-Test SoC is to establish a transmission loop consisting of different transmitter-receiver pairs. Such a transmitter-receiver pair is created by a random configuration of a communication IP containing a receiver and a transmitter module, e.g., an I2S, I2C or DMX CIM. Figure 9.4 shows the setup of the Self-Test SoC that systematically verifies different IP and driver configurations.

The receiver of the IP is directly connected to its transmitter via an HW connection. The driver, however, carries out the task of sending *transmit_ <device>(len, *data)* and reading *receive_ <device>(len, *data)* the data. Accordingly, data packages are thus forwarded through the loop across different IPs. The original data *data_ in* fed into this loop must therefore return intact

at the end of the loop. The correctness of the generation framework is not guaranteed if the data is corrupted.

Conclusively, this simple setup verifies various IPs, IP configurations, and driver implementation variants, improving the quality of the generation framework. So far, more than 50 different variants of the Self-Test SoC have been generated and verified. On average, a loop was composed of approximately 6 IPs.

9.4.2 Safety Demonstrator - Human-Robot Interaction

Safety is a significant issue whenever machines and humans interact with each other. The demonstrator[62] in Figure 9.5 showcases an example of industrial human-robot interaction. The demonstrator depicts an industrial application where a human and a robot jointly work on the same project. The demonstrator's task both are working on is a drawing on a LED panel. The demonstrator is mainly designed to prove the functionality of automatically generated safety patterns ensuring that humans are not exposed to any danger from the robot at any time.

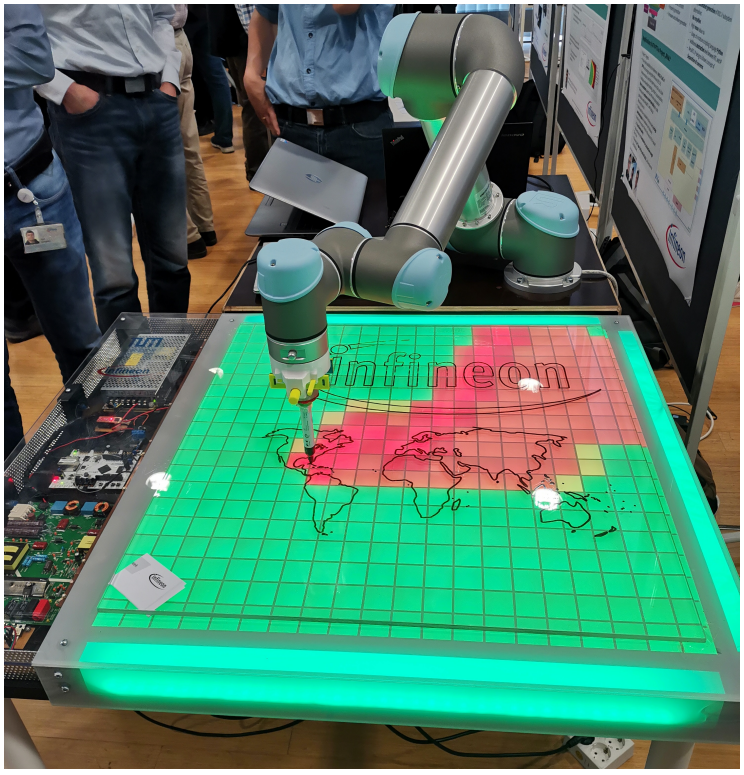


Figure 9.5: *SAFE4I Demonstrator: Cooperative drawing through safe human-robot interaction.*

Before discussing the applicability of automatically generated safety patterns and the achieved results, the demonstrator is explained in detail. The implementation realizes an industrial application that maps a robot arm's position onto a LED matrix to visualize the motion sequences. The demonstrator is divided into two main components: A robot arm with a robot control unit[63] and the light table control unit.

[62] The presented demonstrator has been partly developed in the SAFE4I project, which is funded by the German ministry of education and research (BMBF) (reference number: 01IS17032A) [3]

[63] The robot control unit is developed by the "Forschungszentrum fuer Informatik" FZI, a partner of the SAFE4I project.

The robot's control unit has the task of capturing and sending the near future position (considering the robot arm motion) of the robot arm to the light table control unit. From this data, the LED matrix of the table is colored to highlight the dangerous areas (red) and the safe areas (green) to avoid risks to people and machines.

The generation framework has been used to build the light table control unit. The primary task of this control unit is to receive the coordinates via the DMX interface, process the data and drive the LED matrix via a Serial Peripheral Interface (SPI). The degree of automation is quantified by comparing the number of manually implemented SLOCs with the automatically generated SLOCs. The result of this evaluation, which focuses particularly on the degree of automation for various safety patterns, is illustrated in Table 9.1. The framework achieves an automation rate of 85.3% when building the demonstrator. The share of safety patterns within the entire code is 10.8%.

Table 9.1: *Evaluation of the automatically generated FW safety mechanisms in the light table control system.*

	Number of manual implemented SLOCs	Number of automatic generated SLOCs
Safe Register accesses	0	63
Watchdog	0	26
Program Flow Monitor	0	51
Current Sensor	34	0
Error Handler	23	43
Total Application	327	1891

The safety measures used in the demonstrator (detailed in Section 8.2) are watchdog, PFM, safe register accesses and a current sensor. The result shows that the framework's safety measures do not require additional manual implementation effort. The generation approach is not used for the current sensor as its behavior is very application specific and, therefore, cannot be modeled in an abstract and reusable way. Also, the application-specific error response of the error handler is partially left to the designer. Overall, a substantial degree of automation of 76.3% is achieved for all safety-relevant SLOCs.

9.4.3 ML-based Demonstrator - Location Detection

The demonstrator[64] in Figure 9.6 successfully uses the framework to create the firmware for an ML-based location detection application based on audio information. As with the previous demonstrator, the basic setup, including the LED matrix table, remains the same. However, four microphones are attached to the table corners recording an acoustic knock on the table.

The LED matrix highlights the knocking spot after deriving the exact position from the four audio records. Two different approaches carry out the prediction of the position. First, a conventional method based on the microphones' synchronicity and arrival time difference. Second, the location prediction is performed via signal processing using machine learning.

[64]This demonstrator has been partially funded by the German Federal Ministry of Education and Research (BMBF) within the project Scale4Edge under contract no. 16ME0127 [6]

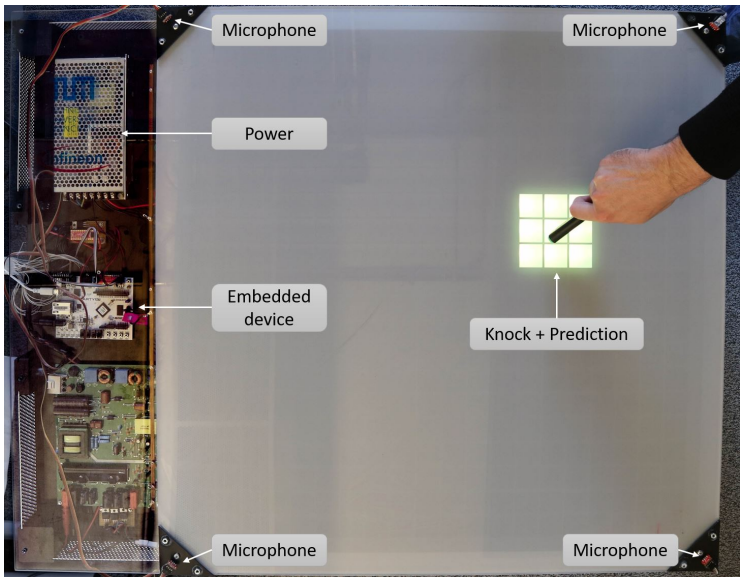


Figure 9.6: *Scale4Edge Demonstrator: 2D automatic location detection based on audio information.*

With the model-driven firmware generation approach, the firmware can be customized individually for each of the two approaches. The demonstrator features the following modules that are entirely handled by the generation framework: I2S interfaces for the microphones, SPI interface to control the LED matrix, I/Os and a general purpose timer. In addition, a UART interface is provided, which is applied in the training phase of the ML-based approach. The UART transmits the audio samples (training data) to the machine that carries out the training of the ML model.

Chapter 10

Summary and Conclusion

This work presents a new methodology to develop embedded systems following model-driven architecture. Rather than considering models as construction blueprints, the established framework points out that models are one solution to today's design challenges. The presented work describes the infrastructure for embedded software metamodeling. It introduces a novel platform-independent model for embedded software design, a generator for code generators, and concepts beyond functional modeling, such as safety transformations and optimization techniques. Along with the hardware generation framework presented in Section 3.2.2, the embedded software framework forms a complete framework for the generation of embedded systems.

Compared to previous generator approaches, this work reduced the effort required to create IP code generators. The work proposed an industry-strength IP code generator framework capable of building the device's hardware and the complete embedded software stack. So, the framework targets different design aspects, such as HW, FW, and the HW/FW interfaces. This holistic consideration reduces the susceptibility to implementation errors by preventing requirements and design inconsistencies. It enables self-adaptive and correct-by-construction implementations of HW, FW and the HW/FW interfaces, which solves the challenges of FW integration. This holistic generation methodology is a major contribution since existing tools rely on pure hardware or firmware generators, which mostly require manual adjustments.

Another contribution is the embedded software metamodel, which formally describes the structure and behavior of embedded software. As a key benefit, the generator addresses not only the skeletons (structure) of the embedded software but also the full functionality (behavior). The generation flow is divided into an IP-specific generator frontend and a generic generator backend. In the frontend, model transformations map the IP specification to an instance of the formal descriptions of the embedded software and hardware. In the generator backend, transformations are applied before feeding the model to code generators. This holistic approach and the tight integration of FW and HW flow ensure the generation of uniform firmware tailored to the hardware. So, the flow maintains a consistent functional interface even if HW details change.

Table 10.1 compares traditional design methodologies with the one presented in this thesis. It highlights the design effort through source lines of code that are required to develop the embedded software of an SPI component. Manually implemented device drivers cover only a single combination of requirements. They are tailored to a specific peripheral device configuration and follow a fixed driver design and target language. Developing code with conditional directives handles more IP configurations but becomes confusingly complex with increasing options. Generation by

traditional templates with a direct conversion from specification to target code can handle all peripheral variants. However, it lacks an intermediate abstract layer to support different driver styles and target languages. The described framework decouples the generation flow into different abstract layers. This abstraction supports the generation of various peripheral configurations, driver variants and languages. Thus, the generator implementation effort is low and comparable to the manual coding effort of a single variant.

Table 10.1: *Analysis of various options for implementing the embedded software of an SPI component.*

	IP requirement combinations	Driver Style combinations	Languages	Effort (SLOC)	Options / Total variants	Remarks
Manual Coding	1	1	1	435	0 / 1	Single Variant Frequent adaptation necessary Error prone
Conditional Directives	6,55E04	1	1	1357	14 / 6,55E04	Limited number of variants Code is hard to read
Template	2,15E09	22	1	1312	21 / 4,72E10	Hard to debug IP dependent transformation Single target platform
MDA	2,15E09	3564	2	489	27 / 1,53E13	IP independent transformation Trade-off analysis Multiple target platforms (languages)

The framework is ideally suited for building embedded software for recurring memory-mapped IPs in SoCs, such as SPI, UART, I2C, Timer, and PIC. The designer only needs to implement the generator frontend of the IP utilizing an intuitive embedded software DSL in Python. The result of the generator frontend is a functional embedded software model conforming to the IP specification. It is further fed into the generic generator backend that supports customizable cross-IP transformations to realize different design decisions. These transformations do not change the actual functionality of the design but perform design extension or implementation-specific translations to:

- generate memory-layout-dependent hardware accesses following different access methodologies.
- build different device driver design alternatives such as generic or specific driver implementations.
- build different I/O concepts, e.g., interrupt, blocking or non-blocking.
- automatically integrate safety measures into the design.

Therefore, the framework's main advantage is the ability to generate a large number of different design alternatives resulting from diverse design decisions. The effort required to support an IP and generate all these variants is minimized because the generation flow is split into an IP-specific frontend generator and a generic backend generator. So a designer only needs to implement

the frontend and can reuse the backend. Table 10.2 provides an overview of the generators' capabilities and illustrates the design effort for the proposed framework. For example, the UART metamodel contains 14 configuration options (just taking into account device requirements). This enables the configuration of approximately 9.43 million different combinations. In order to reach this flexibility, the templates require only 302 SLOC (IP FW template) and 1480 SLOC (IP HW template). The average generated code for a UART is 387 SLOC (generic driver design) and 18.7k SLOC (IP core). When considering all components, a comparable effort (in terms of SLOC) can be observed between generator template implementation and manual programming in C. However, the significant advantage of the presented generator approach becomes evident with the increasing number of realized design variants and projects. It already pays off after the second deployment.

Table 10.2: *Evaluation of generator templates and generated code of different peripheral components*

Abstract Peripheral Model			Generator Template			Generated Files (4 modules each IP)					
Peripheral	Configuration options	Realizable Variants	Design Template SLOC	Driver Template SLOC	Register Interface SLOC	No. Driver Functions	Device specific driver SLOC	Device generic driver SLOC	HAL SLOC	Peripheral Core (HW) SLOC	Register Interface (HW) SLOC
UART	14	9,43E06	1480	302	108	8	760	387	446	18727	1898
DMX	13	8,39E06	1327	377	128	8	859	425	886	15603	2156
I2S	12	1,05E06	907	489	141	11	549	317	1014	9253	2591
SPI	19	2,15E09	1138	471	170	11	820	435	907	10858	2090
PAD	10	1,18E06	412	440	44	6	548	299	122	2056	1116
Timer	9	4,92E05	625	378	49	13	1260	378	365	4828	3693
PIC	12	3,69E05	788	284	72	3	103	103	757	4878	1750

The proposed framework produces not only a complete compilable embedded software stack but also a synthesizable hardware design for FPGA. The generated embedded software code can thus be compiled and simulated directly on the FPGA, offering many advantages and new opportunities. First, the compilability, but also the functional correctness of the designs, is automatically validated. Second, design metrics, such as power, memory consumption or hardware area, are automatically evaluated. This, in turn, helps the user of the proposed framework to either choose between different design alternatives based on the design cost or rely on an automatic optimization step that selects the correct alternative. In this way, certain parameters can be hidden from the user, and automatic optimization can determine the best configuration based on a predefined cost function. This work successfully demonstrated two optimization concepts: memory layout optimization and compiler flag optimization.

Bibliography

[1] Cars are made of code. URL https://www.nxp.com/company/blog/cars-are-made-of-code:BL-CARS-MADE-CODE.

[2] Functional thinking. URL https://www.oreilly.com/library/view/functional-thinking/9781449365509/ch01.html.

[3] URL https://www.edacentrum.de/safe4i/.

[4] Ieee standard for ip-xact, standard structure for packaging, integrating, and reusing ip within tool flows. *IEEE Std 1685-2014 (Revision of IEEE Std 1685-2009)*, pages 1–510, Sep. 2014. doi: 10.1109/IEEESTD.2014.6898803.

[5] Csrcompiler™, Jul 2020. URL https://semifore.com/csrcompiler/. [Online; accessed 28-August-2022].

[6] Zuse-scale4edge, Jun 2020. URL https://www.elektronikforschung.de/projekte/zuse-scale4edge.

[7] Andrea Acquaviva, Nicola Bombieri, Franco Fummi, and Sara Vinco. Semi-automatic generation of device drivers for rapid embedded platform development. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(9):1293–1306, 2013.

[8] Paul Adamczyk. The anthology of the finite state machine design patterns. In *The 10th Conference on Pattern Languages of Programs*, 2003.

[9] Deniz Akdur, Vahid Garousi, and Onur Demirörs. A survey on modeling and model-driven engineering practices in the embedded software industry. *Journal of Systems Architecture*, 91:62–82, 2018.

[10] Pablo Oliveira Antonino, Thorsten Keuler, and Elisa Yumi Nakagawa. Towards an approach to represent safety patterns. In *Proceedings*, 2012.

[11] Enterprise Architect. Sparx systems, 2010.

[12] Ashraf Armoush, Falk Salewski, Stefan Kowalewski, et al. Design pattern representation for safety-critical embedded systems. *Journal of Software Engineering and Applications*, 2(01): 1, 2009.

[13] Divya Arora, Srivaths Ravi, Anand Raghunathan, and Niraj K Jha. Secure embedded processing through hardware-assisted run-time monitoring. In *Design, Automation and Test in Europe*, pages 178–183. IEEE, 2005.

[14] Divya Arora, Srivaths Ravi, Anand Raghunathan, and Niraj K Jha. Hardware-assisted runtime monitoring for secure program execution on embedded processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(12):1295–1308, 2006.

[15] Inc. Arteris. Soc amp; hardware-software interface (hsi) development product. URL https://www.arteris.com/soc-hardware-software-interface-hsi-development-product.

[16] Krste Asanovic and Andrew Waterman. The risc-v instruction set manual. In *Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified*, volume 2. RISC-V Foundation, 2019.

[17] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 4, 2016.

[18] Amir H Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. A survey on compiler autotuning using machine learning. *ACM Computing Surveys (CSUR)*, 51(5):1–42, 2018.

[19] Motor Industry Software Reliability Association and Motor Industry Software Reliability Association Staff. *MISRA C:2012: Guidelines for the Use of the C Language in Critical Systems*. Motor Industry Research Association, 2013. ISBN 9781906400101.

[20] Rabie Ben Atitallah, Philippe Marquet, Éric Piel, Samy Meftali, Smail Niar, Anne Etien, Jean-Luc Dekeyser, and Pierre Boulet. Gaspard2: from marte to systemc simulation. In *Proceeedings of the DATE'08 workshop on Modeling and Analyzis of Real-Time and Embedded Systems with the MARTE UML profile*, 2008.

[21] AUTOSAR. Overview of functional safety measures in autosarautosar cprelease 4.3.0. https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_EXP_FunctionalSafetyMeasures.pdf, 2017.

[22] AUTOSAR. Specification of watchdog manager autosar cp release 4.3.1. https://www.autosar.org/fileadminuser_upload/standards/classic/4-3/AUTOSAR_SWS_WatchdogManager.pdf, 2017.

[23] GbR AUTOSAR. Specification of i/o hardware abstraction. *Website. http://www. autosar. org/download/AUTOSAR_SWS_IO_ HWAbstraction. pdf. Version*, 36:117, 2007.

[24] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *DAC Design automation conference 2012*, pages 1212–1221. IEEE, 2012.

[25] Brian Bailey. Firmware skills shortage, Mar 2021. URL https://semiengineering.com/ai-ml-skills-shortage/.

[26] Anupam Bakshi. Idesignspec (tm) don't fear change, embrace it. *Agnisys Inc. May*, 1, 2008.

[27] Sebastian Baltes and Stephan Diehl. Sketches and diagrams in practice. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 530–541, 2014.

[28] N Md Jubair Basha, Salman Abdul Moiz, and Mohammed Rizwanullah. Model based software development: issues & challenges. *Special Issue of International Journal of Computer Science & Informatics (IJCSI), ISSN (PRINT)*, 2(1):2, 2012.

[29] Benoit Baudry, Sudipto Ghosh, Franck Fleurey, Robert France, Yves Le Traon, and Jean-Marie Mottu. Barriers to systematic model transformation testing. *Communications of the ACM*, 53(6):139–143, 2010.

[30] Ron Bell. Introduction to iec 61508. In *Acm international conference proceeding series*, volume 162, pages 3–12. Citeseer, 2006.

[31] E Bendersky. Pycparser c parser and ast generator written in python, 2012.

[32] Jacob Beningo. Concepts for developing portable firmware. In *Reusable Firmware Development*, pages 1–28. Springer, 2017.

[33] Jacob Beningo, Jacob Beningo, and Anglin. *Reusable Firmware Development*. Springer, 2017.

[34] Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2): 171–188, 2005.

[35] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the omg/mda framework. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 273–280. IEEE, 2001.

[36] Vijay Deep Bhatt, Wolfgang Ecker, Volkan Esen, Zhao Han, Daniela Sanchez Lopera, Rituj Patel, Lorenzo Servadei, Sahil Singla, Sven Wenzek, Vijaydeep Yadav, et al. Soc design automation with ml-it's time for research. In *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*, pages 35–36, 2020.

[37] François Bodin, Toru Kisuki, Peter Knijnenburg, Mike O'Boyle, and Erven Rohou. Iterative compilation in a non-linear optimisation space. 1998.

[38] Nicola Bombieri, Franco Fummi, Graziano Pravadelli, and Sara Vinco. Correct-by-construction generation of device drivers based on rtl testbenches. In *2009 Design, Automation & Test in Europe Conference & Exhibition*, pages 1500–1505. IEEE, 2009.

[39] Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk. Generative software-based memory error detection and correction for operating system data structures. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2013.

[40] Gerry Boyd. Executable uml: Diagrams for the future. *available from Internet< http://www. devx. com/enterprise/Article/10717>(30 December 2004)*, 2003.

[41] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-driven software engineering in practice. *Synthesis lectures on software engineering*, 3(1):1–207, 2017.

[42] Travis Breaux and Jennifer Moritz. The 2021 software developer shortage is coming. *Communications of the ACM*, 64(7):39–41, 2021.

[43] Etienne Brosse, Imran R Quadri, Andrey Sadovykh, Frank Ieromnimon, Dimitrios Kritharidis, Rafael Catrou, and Michel Sarlotte. Enosys fp7 eu project: An integrated modeling and synthesis flow for embedded systems design. In *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–5. IEEE, 2012.

[44] Alan W Brown. Model driven architecture: Principles and practice. *Software and systems modeling*, 3(4):314–327, 2004.

[45] Manfred Broy, Sascha Kirstan, Helmut Krcmar, and Bernhard Schätz. What is the benefit of a model-based design of embedded software systems in the car industry? In *Emerging technologies for the evolution and maintenance of software models*, pages 343–369. IGI global, 2012.

[46] Hugo Bruneliere, Erik Burger, Jordi Cabot, and Manuel Wimmer. A feature-based survey of model view approaches. *Software & Systems Modeling*, 18(3):1931–1952, 2019.

[47] Barrett R Bryant, Jeff Gray, Marjan Mernik, Peter J Clarke, Robert B France, and Gabor Karsai. Challenges and directions in formalizing the semantics of modeling languages. 2011.

[48] Antonio Bucchiarone, Jordi Cabot, Richard F Paige, and Alfonso Pierantonio. Grand challenges in model-driven engineering: an analysis of the state of the research. *Software and Systems Modeling*, 19(1):5–13, 2020.

[49] Ondrej Burkacky, Julia Dragon, and Nikolaus Lehmann. The semiconductor decade: A trillion-dollar industry, Apr 2022. URL https://www.mckinsey.com/industries/semiconductors/our-insights/the-semiconductor-decade-a-trillion-dollar-industry.

[50] Leslie Pérez Cáceres, Federico Pagnozzi, Alberto Franzin, and Thomas Stützle. Automatic configuration of gcc using irace. In *International Conference on Artificial Evolution (Evolution Artificielle)*, pages 202–216. Springer, 2017.

[51] Daniela Cancila, Francois Terrier, Fabien Belmonte, Hubert Dubois, Huascar Espinoza, Sébastien Gérard, and Arnaud Cuccuru. Sophia: a modeling language for model-based safety engineering. In *ACES-MB@ MoDELS*. Citeseer, 2009.

[52] Hui Chen, Guillaume Godet-Bar, Frédéric Rousseau, and Frédéric Pétrot. Me3d: A model-driven methodology expediting embedded device driver development. In *2011 22nd IEEE International Symposium on Rapid System Prototyping*, pages 171–177. IEEE, 2011.

[53] Yang Chen, Shuangde Fang, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Olivier Temam, and Chengyong Wu. Deconstructing iterative optimization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(3):1–30, 2012.

[54] Kiho Choi, Daejin Park, and Jeonghun Cho. Sscfm: Separate signature-based control flow error monitoring for multi-threaded and multi-core environments. *Electronics*, 8(2):166, 2019.

[55] Christopher L Conway and Stephen A Edwards. Ndl: a domain-specific language for device drivers. *ACM Sigplan Notices*, 39(7):30–36, 2004.

[56] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, pages 1–17. USA, 2003.

[57] Alberto Rodrigues Da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43:139–155, 2015.

[58] Roberto de Medeiros, Marcilyanne M Gois, Drausio L Rossi, and Vanderlei Bonato. Designing embedded systems with marte: A pim to psm converter. In *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, pages 303–306. IEEE, 2012.

[59] Jan Decaluwe. Myhdl: a python-based hardware description language. *Linux journal*, 2004 (127):5, 2004.

[60] Marco Di Natale, David Perillo, Francesco Chirico, Andrea Sindico, and Alberto Sangiovanni-Vincentelli. A model-based approach for the synthesis of software to firmware adapters for use with automatically generated components. *Software & Systems Modeling*, 17(1):11–33, 2018.

[61] Francisco Assis Moreira do Nascimento, Marcio FS Oliveira, and Flávio Rech Wagner. A model-driven engineering framework for embedded systems design. *Innovations in Systems and Software Engineering*, 8(1):19–33, 2012.

[62] Gabriel Dos Reis, Jose Daniel Garcia, J Lakos, A Meredith, N Myers, and B Stroustrup. Support for contract based programming in c++. *C++ Standards Committee Working Group ISOCPP*, 2018.

[63] Bruce Powel Douglass. *Doing hard time: developing real-time systems with UML, objects, frameworks, and patterns*, volume 1. Addison-Wesley Professional, 1999.

[64] Wolfgang Ecker and Johannes Schreiner. Introducing model-of-things (mot) and model-of-design (mod) for simpler and more efficient hardware generators. In *2016 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 1–6. IEEE, 2016.

[65] Wolfgang Ecker and Johannes Schreiner. Metamodeling and code generation in the hardware/software interface domain. In *Handbook of hardware/software Codesign*, pages 1051–1091. Springer, 2017.

[66] Wolfgang Ecker, Michael Velten, Leily Zafari, Ajay Goyal, and Wolfgang Mueller. Meta-modeling and code generation-the infineon approach. In *MeCoES-Metamodelling and Code Generation for Embedded Systems: Workshop with ESWEEK*, pages 1–4, 2012.

[67] Wolfgang Ecker, Michael Velten, Leily Zafari, and Ajay Goyal. The metamodeling approach to system level synthesis. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–2. IEEE, 2014.

[68] Wolfgang Ecker, Keerthikumara Devarajegowda, Michael Werner, Zhao Han, and Lorenzo Servadei. Embedded systems' automation following omg's model driven architecture vision. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1301–1306. IEEE, 2019.

[69] Charles Elkan. Using the triangle inequality to accelerate k-means. In *Proceedings of the 20th international conference on Machine Learning (ICML-03)*, pages 147–153, 2003.

[70] Elfatih AB Eltahir and RL Bras. Estimation of the fractional coverage of rainfall in climate models. *Journal of Climate*, 6(4):639–644, 1993.

[71] Embedded. 2019 embedded markets study, Nov 2019. URL https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf.

[72] Elias Fallon. Machine learning in eda: Opportunities and challenges. In *2020 ACM/IEEE 2nd Workshop on Machine Learning for CAD (MLCAD)*, pages 103–103. IEEE, 2020.

[73] Anum Fatima, Shazia Bibi, and Rida Hanif. Comparative study on static code analysis tools for c/c++. In *2018 15th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*, pages 465–469. IEEE, 2018.

[74] Harry D Foster. Why the design productivity gap never happened. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 581–584. IEEE, 2013.

[75] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *Future of Software Engineering (FOSE'07)*, pages 37–54. IEEE, 2007.

[76] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, et al. Milepost gcc: Machine learning enabled self-tuning compiler. *International journal of parallel programming*, 39(3):296–327, 2011.

[77] Daniel D Gajski, Allen C-H Wu, Viraphol Chaiyakul, Shojiro Mori, Tom Nukiyama, and Pierre Bricaud. Embedded tutorial: essential issues for ip reuse. In *Proceedings of the 2000 Asia and South Pacific Design Automation Conference*, pages 37–42, 2000.

[78] GNU GCC. Gcc, the gnu compiler collection. *URL: https://gcc. gnu. org*, 2021.

[79] Kyriakos Georgiou, Craig Blackmore, Samuel Xavier-de Souza, and Kerstin Eder. Less is more: Exploiting the standard compiler optimization levels for better performance and energy consumption. In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*, pages 35–42, 2018.

[80] Sébastien Gérard and Bran Selic. The uml–marte standardized profile. *IFAC Proceedings Volumes*, 41(2):6909–6913, 2008.

[81] Mario Gleirscher and Stefan Kugele. Assurance of system safety: A survey of design and argument patterns. *arXiv preprint arXiv:1902.05537*, 2019.

[82] Cesar Gonzalez Perez and Brian Henderson-Sellers. *Metamodelling for software engineering.* John Wiley and Sons, 2008.

[83] M Gouda. Cmsis-rtos an api interface standard for real-time operating systems. In *ARM Technology Symposia*, 2012.

[84] Per Haglund, Oskar Mencer, Wayne Luk, and Benjamin Tai. Pyhdl: Hardware scripting with python. In *Engineering of Reconfigurable Systems and Algorithms*, pages 288–291, 2003.

[85] Zhao Han, Keerthikumara Devarajegowda, Michael Werner, and Wolfgang Ecker. Towards a python-based one language ecosystem for embedded systems automation. In *2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*, pages 1–7. IEEE, 2019.

[86] Zhao Han, Shahzaib Qazi, Michael Werner, Keerthikumara Devarajegowda, and Wolfgang Ecker. On self-verifying dsl generation for embedded systems automation. In *MBMV 2021; 24th Workshop*, pages 1–7. VDE, 2021.

[87] Robert S Hanmer. *Patterns for fault tolerant software.* John Wiley & Sons, 2013.

[88] George T Heineman and William T Councill. Component-based software engineering. *Putting the pieces together, addison-westley*, 5, 2001.

[89] Fernando Herrera, Julio Medina, and Eugenio Villar. Modeling hardware/software embedded systems with uml/marte: a single-source design approach. In *Handbook of Hardware/-Software Codesign*, pages 141–185. Springer, 2017.

[90] Geoffrey E Hinton, Terrence Joseph Sejnowski, Tomaso A Poggio, et al. *Unsupervised learning: foundations of neural computation.* MIT press, 1999.

[91] J Hruska. As chip design costs skyrocket, 3nm process node is in jeopardy. *ExtremeTech https://www. extremetech. com/computing/272096-3nm-process-node (22 June 2018)*, 2018.

[92] Guyue Huang, Jingbo Hu, Yifan He, Jialong Liu, Mingyuan Ma, Zhaoyang Shen, Juejian Wu, Yuanfan Xu, Hengrui Zhang, Kai Zhong, et al. Machine learning for electronic design automation: A survey. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 26(5):1–46, 2021.

[93] Lars Huning and Elke Pulvermueller. Automatic code generation of safety mechanisms in model-driven development. *Electronics*, 10(24):3150, 2021.

[94] Lars Huning, Padma Iyenghar, and Elke Pulvermüller. Uml specification and transformation of safety features for memory protection. In *ENASE*, pages 281–288, 2019.

[95] Lars Huning, Padma Iyenghar, and Elke Pulvermüller. Uml-based model-driven code generation of error detection mechanisms. *ICSEA 2020*, page 108, 2020.

[96] John Hutchinson, Mark Rouncefield, and Jon Whittle. Model-driven engineering practices in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 633–642, 2011.

[97] Accellera Systems Initiative. SystemRDL 2.0: Register Description Language. https://www.accellera.org/downloads/standards/systemrdl, 2018. [Online; accessed 22-August-2019].

[98] ISO, CD. Road vehicles — functional safety — part 6: Product development at the software level. *International Standard ISO/FDIS*, 2, 2018.

[99] ISO/IEC 9899:1999. Programming languages — c. Standard, American National Standards Institute, December 1999.

[100] Ahmed A Jerraya and Wayne Wolf. Hardware/software interface codesign for embedded systems. *Computer*, 38(2):63–69, 2005.

[101] M Tim Jones. Optimization in gcc. *Linux journal*, 2005(131):11, 2005.

[102] Ademir AC Júnior, Sanjay Misra, and Michel S Soares. A systematic mapping study on software architectures description based on iso/iec/ieee 42010: 2011. In *International Conference on Computational Science and Its Applications*, pages 17–30. Springer, 2019.

[103] Endri Kaja, Nicolas Gerlin, Mounika Vaddeboina, Luis Rivas, Sebastian Prebeck, Zhao Han, Keerthikumara Devarajegowda, and Wolfgang Ecker. Towards fault simulation at mixed register-transfer/gate-level models. In *2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–6. IEEE, 2021.

[104] Endri Kaja, Nicolas Ojeda Leon, Michael Werner, Bogdan Andrei-Tabacaru, Keerthikumara Devarajegowda, and Wolfgang Ecker. Extending verilator to enable fault simulation. In *MBMV 2021; 24th Workshop*, pages 1–6. VDE, 2021.

[105] Antti Kamppi, Lauri Matilainen, Joni-Matti Määttä, Erno Salminen, and Timo D Hämäläinen. Extending ip-xact to embedded system hw/sw integration. In *2013 International Symposium on System on Chip (SoC)*, pages 1–8. IEEE, 2013.

[106] Antti Kamppi, Esko Pekkarinen, Janne Virtanen, Joni-Matti Määttä, Juho Järvinen, Lauri Matilainen, Mikko Teuho, and Timo D Hämäläinen. Kactus2: A graphical eda tool built on the ip-xact standard. *Journal of Open Source Software*, 2(13):151, 2017.

[107] Sungwon Kang, Hyunho Kim, Jongmoon Baik, Hojin Choi, and Changsup Keum. Transformation rules for synthesis of uml activity diagram from scenario-based specification. In *2010 IEEE 34th Annual Computer Software and Applications Conference*, pages 431–436. IEEE, 2010.

[108] Gerti Kappel, Elisabeth Kapsammer, Horst Kargl, Gerhard Kramler, Thomas Reiter, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. Lifting metamodels to ontologies: A step to the semantic integration of modeling languages. In *International Conference on Model Driven Engineering Languages and Systems*, pages 528–542. Springer, 2006.

[109] Tetsuro Katayama, Keizo Saisho, and Akira Fukuda. Prototype of the device driver generation system for unix-like operating systems. In *Proceedings International Symposium on Principles of Software Evolution*, pages 302–310. IEEE, 2000.

[110] Matti Käyrä and Timo D Hämäläinen. A survey on system-on-a-chip design using chisel hw construction language. In *IECON 2021–47th Annual Conference of the IEEE Industrial Electronics Society*, pages 1–6. IEEE, 2021.

[111] Brian W Kernighan and Dennis M Ritchie. *The C programming language*. 2006.

[112] Hagbae Kim and Kang G Shin. Evaluation of fault tolerance latency from real-time application's perspectives. *IEEE Transactions on computers*, 49(1):55–64, 2000.

[113] Anneke Kleppe. *Software language engineering: creating domain-specific languages using metamodels*. Pearson Education, 2008.

[114] M. Kooli and G. Di Natale. A survey on simulation-based fault injection tools for complex systems. In *2014 9th IEEE International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–6, 2014. doi: 10.1109/DTIS.2014.6850649.

[115] Israel Koren and C Mani Krishna. *Fault-tolerant systems*. Morgan Kaufmann, 2020.

[116] Jeff Kramer. Is abstraction the key to computing? *Communications of the ACM*, 50(4): 36–42, 2007.

[117] John Krogstie. Modelling of the people, by the people, for the people. In *Conceptual modelling in information systems engineering*, pages 305–318. Springer, 2007.

[118] Wido Kruijtzer, Pieter Van Der Wolf, Erwin De Kock, Jan Stuyt, Wolfgang Ecker, Albrecht Mayer, Serge Hustin, Christophe Amerijckx, Serge De Paoli, and Emmanuel Vaumorin. Industrial ip integration flows based on ip-xact standards. In *2008 Design, Automation and Test in Europe*, pages 32–37. IEEE, 2008.

[119] Dirk Kuschnerus, Felix Bruns, Attila Bilgic, and Thomas Musch. A uml profile for the development of iec 61508 compliant embedded software. In *Embedded Real Time Software and Systems (ERTS2012)*, 2012.

[120] LINAGORA Research Labs. What is MDA? Why concerns BPMN? https://research.linagora.com/pages/viewpage.action?pageId=3639295, 2021. [Online; Accessed 7 August 2022]].

[121] Agnes Lanusse, Yann Tanguy, Huascar Espinoza, Chokri Mraidha, Sebastien Gerard, Patrick Tessier, Remi Schnekenburger, Hubert Dubois, and François Terrier. Papyrus uml: an

open source toolset for mda. In *Proc. of the Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009)*, pages 1–4. Citeseer, 2009.

[122] Christian Leber. Efficient hardware for low latency applications. 2012.

[123] L Lennis and José Aedo. Generation of efficient embedded c code from uml/marte models. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*, page 1. The Steering Committee of The World Congress in Computer Science, Computer . . . , 2013.

[124] Patrick Leserf, Pierre de Saqui-Sannes, and Jérôme Hugues. Trade-off analysis for sysml models using decision points and csps. *Software and Systems Modeling*, 18(6):3265–3281, 2019.

[125] Patrick S. Li, Adam M. Izraelevitz, and Jonathan Bachrach. Specification for the firrtl language. Technical Report UCB/EECS-2016-9, EECS Department, University of California, Berkeley, Feb 2016. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-9.html.

[126] Grischa Liebel, Nadja Marko, Matthias Tichy, Andrea Leitner, and Jörgen Hansson. Assessing the state-of-practice of model-based engineering in the embedded systems domain. In *International conference on model driven engineering languages and systems*, pages 166–182. Springer, 2014.

[127] Grant Likely and Josh Boyer. A symphony of flavours: Using the device tree to describe embedded hardware. In *Proceedings of the Linux Symposium*, volume 2, pages 27–37, 2008.

[128] Edson Lisboa, Luciano Silva, Igino Chaves, Thiago Lima, and Edna Barros. A design flow based on a domain specific language to concurrent development of device drivers and device controller simulation models. In *Proceedings of th 12th International Workshop on Software and Compilers for Embedded Systems*, pages 53–60, 2009.

[129] S. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 1982.

[130] Derek Lockhart, Gary Zibrat, and Christopher Batten. Pymtl: A unified framework for vertically integrated computer architecture research. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 280–292. IEEE, 2014.

[131] Shourong Lu and Wolfgang A Halang. A uml profile to model safety-critical embedded real-time control systems. In *Contributions to Ubiquitous Computing*, pages 197–218. Springer, 2007.

[132] Azad M Madni and Michael Sievers. Model-based systems engineering: Motivation, current status, and research opportunities. *Systems Engineering*, 21(3):172–190, 2018.

[133] Mohammad Maghsoudloo, Hamid R. Zarandi, Saadat Pour Mozafari, and Navid Khoshavi. Soft error detection technique in multi-threaded architectures using control-flow monitoring.

Proceedings - 2011 14th Euromicro Conference on Digital System Design: Architectures, Methods and Tools, DSD 2011, pages 789–792, 2011. doi: 10.1109/DSD.2011.104.

[134] UML MARTE. Uml profile for marte: modeling and analysis of real-time embedded systems, 2015.

[135] Luiz GA Martins, Ricardo Nobre, Joao MP Cardoso, Alexandre CB Delbem, and Eduardo Marques. Clustering-based selection for the exploration of compiler optimization sequences. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(1):1–28, 2016.

[136] OMG MDA. Object management group model driven architecture, 2008.

[137] Fabrice Mérillon, Laurent Réveillere, Charles Consel, Renaud Marlet, and Gilles Muller. Devil: An {IDL} for hardware programming. In *Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, 2000.

[138] Bertrand Meyer. Applying'design by contract'. *Computer*, 25(10):40–51, 1992.

[139] Kaisa Miettinen. *Nonlinear multiobjective optimization*, volume 12. Springer Science & Business Media, 2012.

[140] Parastoo Mohagheghi and Vegard Dehlen. Where is the proof?-a review of experiences from applying mde in industry. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 432–443. Springer, 2008.

[141] Maryam I Mukhtar and Bashir S Galadanci. Automatic code generation from uml diagrams: the state-of-the-art. *Science World Journal*, 13(4):47–60, 2018.

[142] Alex Mykyta. Peakrdl: Command-line tool for control/status register automation. https://pypi.org/project/peakrdl, 2020. [Online; accessed 28-August-2022].

[143] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 189–200. IEEE, 2012.

[144] Juniper Networks.

[145] William F Ogilvie, PetoumeAutomatic Tuning of Compiler Optimizations, Pavlos Analysis of their Impactnos, Zheng Wang, and Hugh Leather. Minimizing the cost of iterative compilation with active learning. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 245–256. IEEE, 2017.

[146] N. Oh, P. P. Shirvani, and E. J. McCluskey. Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 51(1):111–122, 2002. doi: 10.1109/24.994926.

[147] Opentitan. Register tool. https://docs.opentitan.org/doc/rm/register_tool/. [Online; accessed 28-August-2022].

[148] Manish Pandey. Machine learning and systems for building the next generation of eda tools. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 411–415. IEEE, 2018.

[149] Marco Panunzio and Tullio Vardanega. On component-based development and high-integrity real-time systems. In *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 79–84. IEEE, 2009.

[150] Marco Panunzio and Tullio Vardanega. A component model for on-board software applications. In *2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 57–64. IEEE, 2010.

[151] Visual Paradigm. Uml class diagram tutorial. *Visual Paradigm*, 2020.

[152] Karthik Pattabiraman, Vinod Grover, and Benjamin G Zorn. Samurai: protecting critical data in unsafe languages. *ACM SIGOPS Operating Systems Review*, 42(4):219–232, 2008.

[153] Lars Patzina, Sven Patzina, Thorsten Piper, and Paul Manns. Model-based generation of run-time monitors for autosar. In *European Conference on Modelling Foundations and Applications*, pages 70–85. Springer, 2013.

[154] Sandeep Pendharkar and Venugopal Kolathur. Ddgen: An automated device driver generation tool for embedded systems. 2009.

[155] Mauro Pezzé and Jochen Wuttke. Model-driven generation of runtime checks for system properties. *International Journal on Software Tools for Technology Transfer*, 18(1):1–19, 2016.

[156] Dmitry Plotnikov, Dmitry Melnik, Mamikon Vardanyan, Ruben Buchatskiy, Roman Zhuykov, and Je-Hyung Lee. Automatic tuning of compiler optimizations and analysis of their impact. *Procedia Computer Science*, 18:1312–1321, 2013.

[157] Héctor Posadas, Pablo Peñil, Alejandro Nicolás, and Eugenio Villar. Automatic synthesis of embedded sw for evaluating physical implementation alternatives from uml/marte models supporting memory space separation. *Microelectronics Journal*, 45(10):1281–1291, 2014.

[158] Laura L Pullum. *Software fault tolerance techniques and implementation*. Artech House, 2001.

[159] Imran Rafiq Quadri, Samy Meftali, and Jean-Luc Dekeyser. Designing dynamically reconfigurable socs: From uml marte models to automatic code generation. In *2010 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 68–75. IEEE, 2010.

[160] Douglas Paulo Bertrand Renaux. Comparative performance evaluation of cmsis-rtos. In *2014 Brazilian Symposium on Computing Systems Engineering*, pages 126–131. IEEE, 2014.

[161] Matthew J Renzelmann and Michael M Swift. Decaf: Moving device drivers to a modern language. 2009.

[162] A Wendell O Rodrigues, Frédéric Guyomarc'h, and Jean-Luc Dekeyser. An mde approach for automatic code generation from uml/marte to opencl. *Computing in Science & Engineering*, 15(1):46–55, 2012.

[163] Peter Rössler, Roland Höller, Christopher Reisner, and Oliver Maischberger. Survey and comparison of digital logic simulators. In *2019 Austrochip Workshop on Microelectronics (Austrochip)*, pages 87–92. IEEE, 2019.

[164] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004. ISBN 0321245628.

[165] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. Automatic device driver synthesis with termite. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 73–86, 2009.

[166] Mehrdad Saadatmand, Antonio Cicchetti, and Mikael Sjödin. Toward model-based trade-off analysis of non-functional requirements. In *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*, pages 142–149. IEEE, 2012.

[167] Soujanna Sarkar, S Shinde, et al. Effective ip reuse for high quality soc design. In *Proceedings 2005 IEEE International SOC Conference*, pages 217–224. IEEE, 2005.

[168] Bernhard Schätz, Alexander Pretschner, Franz Huber, and Jan Philipps. Model-based development of embedded systems. In *International Conference on Object-Oriented Information Systems*, pages 298–311. Springer, 2002.

[169] Gunar Schirner, Andreas Gerstlauer, and Rainer Domer. Automatic generation of hardware dependent software for mpsocs from abstract system specifications. In *2008 Asia and South Pacific Design Automation Conference*, pages 271–276. IEEE, 2008.

[170] Rolf Schmedes, Philipp Ittershagen, and Kim Grüttner. Towards distributed runtime monitoring with c++ contracts. In *Proceedings of the International Conference on Omni-Layer Intelligent Systems*, pages 141–145, 2019.

[171] Johannes Schreiner and Wolfgang Ecker. Digital hardware design based on metamodels and model transformations. In *IFIP/IEEE International Conference on Very Large Scale Integration-System on a Chip*, pages 83–107. Springer, 2016.

[172] Johannes Schreiner, Rainer Findenigy, and Wolfgang Ecker. Design centric modeling of digital hardware. In *2016 IEEE International High Level Design Validation and Test Workshop (HLDVT)*, pages 46–52. IEEE, 2016.

[173] Johannes Schreiner, Felix Willgerodt, and Wolfgang Ecker. A new approach for generating view generators. In *Design and Verification Conference-US*, 2017.

[174] Michael A. Schuette and John Paul Shen. Processor control flow monitoring using signatured instruction streams. *IEEE Transactions on Computers*, (3):264–276, 1987.

[175] Roger S Scowen. Generic base standards. In *Proceedings 1993 Software Engineering Standards Symposium*, pages 25–34. IEEE, 1993.

[176] Bran Selic. The pragmatics of model-driven development. *IEEE software*, 20(5):19–25, 2003.

[177] Petri Selonen, Kai Koskimies, and Markku Sakkinen. Transformations between uml diagrams. *Journal of Database Management (JDM)*, 14(3):37–55, 2003.

[178] L. Servadei, E. Mosca, M. Werner, V. Esen, R. Wille, and W. Ecker. Combining evolutionary algorithms and deep learning for hardware/software interface optimization. In *2019 ACM/IEEE 1st Workshop on Machine Learning for CAD (MLCAD)*, pages 1–6, 2019. doi: 10.1109/MLCAD48534.2019.9142090.

[179] Lorenzo Servadei, Elena Zennaro, Keerthikumara Devarajegowda, Martin Manzinger, Wolfgang Ecker, and Robert Wille. Accurate cost estimation of memory systems inspired by machine learning for computer vision. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1277–1280. IEEE, 2019.

[180] Lorenzo Servadei, Elena Zennaro, Tobias Fritz, Keerthikumara Devarajegowda, Wolfgang Ecker, and Robert Wille. Using machine learning for predicting area and firmware metrics of hardware designs from abstract specifications. *Microprocessors and Microsystems*, 71: 102853, 2019.

[181] Lorenzo Servadei, Edoardo Mosca, Elena Zennaro, Keerthikumara Devarajegowda, Michael Werner, Wolfgang Ecker, and Robert Wille. Accurate cost estimation of memory systems utilizing machine learning and solutions from computer vision for design automation. *IEEE Transactions on Computers*, 69(6):856–867, 2020.

[182] Lorenzo Servadei, Jiapeng Zheng, José Arjona-Medina, Michael Werner, Volkan Esen, Sepp Hochreiter, Wolfgang Ecker, and Robert Wille. Cost optimization at early stages of design using deep reinforcement learning. In *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*, pages 37–42, 2020.

[183] Lorenzo Servadei, Jin Hwa Lee, José A Arjona Medina, Michael Werner, Sepp Hochreiter, Wolfgang Ecker, and Robert Wille. Deep reinforcement learning for optimization at early design stages. *IEEE Design & Test*, 2022.

[184] John Paul Shen. On-line monitoring using signatured instruction streams. In *Proc. IEEE International Test Conference, Oct. 1983*, pages 275–282, 1983.

[185] Yashwant Singh and Manu Sood. Model driven architecture: A perspective. In *2009 IEEE International Advance Computing Conference*, pages 1644–1652. IEEE, 2009.

[186] Wilson Snyder, Paul Wasson, and Duane Galbi. Verilator. *Direct search methods: then and now*, 2007.

[187] AMBA Specification. Rev. 2.0. *ARM Limited*, 1999.

[188] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. 3 metamodelling. In *Dagstuhl Workshop on Model-Based Engineering of Embedded Real-Time Systems*, pages 57–76. Springer, 2007.

[189] Herbert Stachowiak. General model theory. *Springer*, 1973.

[190] Rafael Stahl, Daniel Mueller-Gritschneder, and Ulf Schlichtmann. Driver generation for iot nodes with optimization of the hardware/software interface. *IEEE Embedded Systems Letters*, 12(2):66–69, 2019.

[191] Thomas Stahl, Markus Völter, and Krzysztof Czarnecki. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, Inc., 2006.

[192] EBNF Syntaxt Specification Standard. Ebnf: Iso/iec 14977: 1996 (e). *URL http://www. cl. cam. ac. uk/mgk25/iso-14977. pdf*, 70, 1996.

[193] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[194] Detlef Streitferdt, Georg Wendt, Philipp Nenninger, Alexander Nyßen, and Horst Lichter. Model driven development challenges in the automation domain. In *2008 32nd Annual IEEE International Computer Software and Applications Conference*, pages 1372–1375. IEEE, 2008.

[195] Jun Sun, Wanghong Yuan, Mahesh Kallahalla, and Nayeem Islam. Hail: a language for easy and correct device access. In *Proceedings of the 5th ACM international conference on Embedded software*, pages 1–9, 2005.

[196] Rickard Svenningsson, Jonny Vinter, Henrik Eriksson, and Martin Törngren. Modifi: a model-implemented fault injection tool. In *International Conference on Computer Safety, Reliability, and Security*, pages 210–222. Springer, 2010.

[197] Eugene Syriani, Lechanceux Luhunu, and Houari Sahraoui. Systematic mapping study of template-based code generation. *Computer Languages, Systems & Structures*, 52:43–62, 2018.

[198] Program Verification Systems. PVS-Studio Analyzer. https://pvs-studio.com/en/m/, 2021. [Online; accessed 20-April-2021].

[199] Bogdan-Andrei Tabacaru, Moomen Chaari, Wolfgang Ecker, Thomas Kruse, and Cristiano Novello. Fault-effect analysis on system-level hardware modeling using virtual prototypes. In *2016 Forum on Specification and Design Languages (FDL)*, pages 1–7. IEEE, 2016.

[200] Bogdan-Andrei Tabacaru, Moomen Chaari, Wolfgang Ecker, Thomas Kruse, and Cristiano Novello. Speeding up safety verification by fault abstraction and simulation to transaction level. In *2016 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 1–6. IEEE, 2016.

[201] Kenji Taguchi. Meta modeling approach to safety standard for consumer devices. In *Seminar on Systems Assurance & Safety for Consumer Devices*, 2011.

[202] Ishitani Taichi. Rggen. https://www.librecores.org/taichi-ishitani/rggen, 2015. [Online; accessed 28-August-2022].

[203] Julien Tanguy, Jean-Luc Béchennec, Mikaël Briday, Sébastien Dubé, and Olivier H Roux. Device driver synthesis for embedded systems. In *2013 IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*, pages 1–8. IEEE, 2013.

[204] Lydie Terras, Yannick Teglia, Michel Agoyan, and Régis Leveugle. Taking into account indirect jumps or calls in continuous control-flow checking. In *2016 11th International Design & Test Symposium (IDT)*, pages 125–130. IEEE, 2016.

[205] Robert L Thorndike. Who belongs in the family. In *Psychometrika*. Citeseer, 1953.

[206] Nicholas H Tollervey. *Programming with MicroPython: embedded programming with microcontrollers and Python*. " O'Reilly Media, Inc.", 2017.

[207] Marco Torchiano, Federico Tomassetti, Filippo Ricca, Alessandro Tiso, and Gianna Reggio. Preliminary findings from a survey on the md state of the practice. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 372–375. IEEE, 2011.

[208] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I August. Compiler optimization-space exploration. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 204–215. IEEE, 2003.

[209] Raphael Fonte Boa Trindade, Lukas Bulwahn, and Christoph Ainhauser. Automatically generated safety mechanisms from semi-formal software safety requirements. In *International Conference on Computer Safety, Reliability, and Security*, pages 278–293. Springer, 2014.

[210] Dimitri Van Heesch. Doxygen: Source code documentation generator tool. *URL: http://www. doxygen. org*, 2008.

[211] Pham Van Huong and Nguyen Ngoc Binh. An approach to design embedded systems by multi-objective optimization. In *The 2012 International Conference on Advanced Technologies for Communications*, pages 165–169. IEEE, 2012.

[212] Jens Vankeirsbilck, Niels Penneman, Hans Hallez, and Jeroen Boydens. Random additive signature monitoring for control flow error detection. *IEEE transactions on Reliability*, 66 (4):1178–1192, 2017.

[213] Jose Luis de la Vara and Rajwinder Kaur Panesar-Walawege. Safetymet: A metamodel for safety standards. In *International Conference on Model Driven Engineering Languages and Systems*, pages 69–86. Springer, 2013.

[214] Jorgiano Vidal, Florent De Lamotte, Guy Gogniat, Philippe Soulard, and Jean-Philippe Diguet. A co-design approach for embedded system modeling and code generation with uml and marte. In *2009 Design, Automation & Test in Europe Conference & Exhibition*, pages 226–231. IEEE, 2009.

[215] Markus Voelter, Christian Salzmann, and Michael Kircher. Model driven software development in the context of embedded component infrastructures. In *Component-Based Software Development for Embedded Systems*, pages 143–163. Springer, 2005.

[216] Markus Voelter, Daniel Ratiu, Bernd Kolb, and Bernhard Schaetz. mbeddr: Instantiating a language workbench in the embedded software domain. *Automated Software Engineering*, 20(3):339–390, 2013.

[217] Markus Voelter, Arie van Deursen, Bernd Kolb, and Stephan Eberle. Using c language extensions for developing embedded software: A case study. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 655–674, 2015.

[218] Shaojie Wang, Sharad Malik, and Reinaldo A Bergamaschi. Modeling and integration of peripheral devices in embedded systems. In *2003 Design, Automation and Test in Europe Conference and Exhibition*, pages 136–141. IEEE, 2003.

[219] Nancy J Warter and Wen-Mei W Hwu. Compiler-assisted signature monitoring. *Coordinated Science Laboratory Report no. UILU-ENG-90-2236, CRHC-90-6*, 1990.

[220] Mario Werner, Erich Wenger, and Stefan Mangard. Protecting the control flow of embedded processors against fault attacks. In *International Conference on Smart Card Research and Advanced Applications*, pages 161–176. Springer, 2015.

[221] Michael Werner, Keerthikumara Devarajegowda, Moomen Chaari, and Wolfgang Ecker. Increasing soft error resilience by software. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–4. IEEE, 2019.

[222] Michael Werner, Andreas Neumeier, and Wolfgang Ecker. A syntax oriented code generation approach for soc design automation. In *Final Workshop Proceedings*, page 33, 2019.

[223] Michael Werner, Lorenzo Servadei, Robert Wille, and Wolfgang Ecker. Automatic compiler optimization on embedded software through k-means clustering. In *2020 ACM/IEEE 2nd Workshop on Machine Learning for CAD (MLCAD)*, pages 157–162. IEEE, 2020.

[224] Michael Werner, Igli Zeraliu, Zhao Han, Sebastian Prebeck, Lorenzo Servardei, and Wolfgang Ecker. Optimized hw/fw generation from an abstract register interface model. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 35–39. IEEE, 2020.

[225] Lea Wittie. Laddie: The language for automated device drivers (ver 1. 2008.

[226] Michael V Woodward and Pieter J Mosterman. Challenges for embedded software development. In *2007 50th Midwest Symposium on Circuits and Systems*, pages 630–633. IEEE, 2007.

[227] Yanxia Wu, Guochang Gu, Shaobin Huang, and Jun Ni. Control flow checking algorithm using soft-basedintra-/inter-block assigned-signature. In *Second International Multi-Symposiums on Computer and Computational Sciences (IMSCCS 2007)*, pages 412–415. IEEE, 2007.

[228] Jisoo Yang, Dave B Minturn, and Frank Hady. When poll is better than interrupt. In *FAST*, volume 12, pages 3–3, 2012.

[229] Benabdallah Ahcene Youcef and Boudour Rachid. A fast prototype for modeling ip cores using in soc with uml marte. *Informatica*, 45(6), 2021.

[230] Elena Zennaro, Lorenzo Servadei, Keerthikumara Devarajegowda, and Wolfgang Ecker. A machine learning approach for area prediction of hardware designs from abstract specifications. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pages 413–420. IEEE, 2018.

[231] Igli Zeraliu. Automated generation of an optimized hal from the register interface models. Master's thesis, TU Kaiserslautern, 2020.

[232] Qing-Li Zhang, Ming-Yuan Zhu, and Shuo-Ying Chen. Automatic generation of device drivers. *ACM SIGPLAN Notices*, 38(6):60–69, 2003.