Technische Universität München
TUM School of Computation, Information and Technology

Model-based Generation of Highly Configurable RTL Designs

Johannes B. Schreiner

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

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

genehmigten Dissertation.

Vorsitz:		Prof. Dr.-Ing. Ralf Brederlow
Prüfer*innen der Dissertation:	1.	Hon.-Prof. Dr.-Ing. Wolfgang Ecker
	2.	Prof. Dr.-Ing. Robert Wille

Die Dissertation wurde am 24.04.2023 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 23.11.2023 angenommen.

Acknowledgements

This research has been made possible by Infineon Technologies AG and the EDA Chair at Technical University of Munich. I would like to express my sincere gratitude for the opportunity to conduct research with them.

Above all else, I would like to thank Prof. Dr.-Ing. Wolfgang Ecker for his constant support, guidance, and valuable inputs – not only throughout my work on this thesis but also during my entire career at Infineon. Despite his busy schedule, he was always available for thoughtful discussions, creative ideas, and valuable feedback. In countless meetings and alignments, he pushed me in the right direction and exhibited incredible patience. I deeply value his ability to motivate and push me by showing true appreciation and enthusiasm for my work and results. If there is one thing that I hope to learn from him and be able to carry forward through my professional life, it is this ability to motivate and drive others by feeling and showing the same honest fascination and appreciation for their work. Without his patience, persistence, and encouragement I would have never been able to come to these results. I am aware of how extraordinary the opportunities that he opens up to me are and I cannot express how thankful I am. I will do my best to make sure that my future contributions at Infineon Technologies AG provide a return on his investment.

Further, I want to thank my mentor Prof. Dr.-Ing. Ulf Schlichtmann. I am very thankful that he decided to mentor me personally and I want to express my deepest gratitude for his feedback and the essential support that helped me finalize this work.

A special thank you goes to all the students who contributed key elements to the development and research conducted as part of this thesis: Felix Willgerodt, Bastian Koppermann, Yurun Wu, Andreas Neumeier, and Vasundhara Raje Gontia. I would also like to thank all Ph.D. candidates who based their work on the foundations set by this thesis and all Infineon employees working in the Infineon Metamodeling Group. They evolved the interpretation of the MDA flow developed in this thesis into what came to be Infineon's MetaX initiative. In this context, I want to particularly mention Keerthikumara Devarajegowda, Stefan Gerstendörfer, Paritosh Kumar Sinha, Andreas Neumeier, and Christian Lück who offered their guidance, support, and honest opinions. The motivation their feedback provided is invaluable to me. I also want to thank Sebastian Prebeck and Michael Werner – their Ph.D. work has inspired, guided, and motivated me. My special thank you for all the insightful and incredibly motivating conversations that allowed me to finalize this thesis.

A big thank you also goes to my line managers Jens Eckardt and Anshuman Anand for supporting me and providing the flexibility I needed to work on and finish this thesis.

Finally, I would like to thank my wife Ronja, my friends, and my family for the support and the distraction they offered me during the more challenging stages of my work on this thesis.

Abstract

The exponential growth in the capabilities of integrated circuits (ICs) has massively increased the complexity and cost of their development. To manage this complexity, a high degree of specialization, tool support, and reuse of intellectual property (so-called IP-reuse) between different integrated circuits are critical. Code generators can automatically provide artifacts for software and hardware development such as synthesis models on Register Transfer Level (RTL) or software code for device firmware. In areas where code generation can be applied, the replacement of manual coding with automated generation has shown significant productivity increases.

The applicability of code generation is limited by the complexity inherent in developing generators. This thesis provides a novel methodology for developing generators for digital hardware that significantly reduces the complexity of generator development and the cost of generator development along with it. The reduced complexity further facilitates the development of generator-based automation for digital design tasks which previously required manual work.

The new methodology is based on the vision of Model Driven Architecture (MDA) which has been developed in the field of software engineering and is successfully applied there. Model Driven Architecture aims to automate the generation of target views through a series of models with different levels of abstraction and Model-to-Model transformations that automatically provide more refined models from more abstract specification models and eventually automatically generate the final view.

This thesis adapts, transfers, and extends the vision of Model Driven Architecture for use in the field of digital hardware design. It identifies key Metamodels and Model-to-Model transformations and implements an end-to-end generation flow for digital designs. The framework developed in the context of this thesis puts particular focus on Metamodel-based automation to reduce the manual effort required to develop generators.

The new framework enables a shift from the development of static instances of digital hardware in hardware description languages such as SystemVerilog and VHDL to the development of highly configurable generators in a general-purpose software programming language. This thesis demonstrates how the capabilities of this general-purpose programming language and the MDA-inspired modeling environment can be applied for generator reuse and to solve complex System-on-Chip (SoC) infrastructure problems.

The results of this thesis are the foundation of MetaX, a full chip generation framework that is already used in a productive industrial context to automate chip design, verification, and embedded software development where it is able to reduce the overall development effort significantly.

Contents

List of Figures

List of Listings

1. Introduction

1.1. Growing Complexity of Integrated Circuits

The computer age started with the use of vacuum tubes as switches in calculating machines. The world's first programmable, fully automated computer based on vacuum tubes is the Z3 with its 2,600 tubes. It was developed in 1939. The end of the relatively short history of the development of vacuum tube based computers is marked by the initial delivery of the AN/FSQ-7, the largest and most powerful vacuum tube based computer in 1957. This computer was built from 49,000 vacuum tubes, required more than 2,000 m^2 of floor space, and weighed 275 tons at a power usage of up to three megawatts. These dimensions still make it one of the largest computers ever built – measuring up to today's supercomputers [91, 9, 19].

It was clear that the fundamental boundaries were limiting the evolution of vacuum tube based computers. At the time, it was easy for engineers to design systems with a complexity much higher than what could possibly be built: if those systems had been realized on vacuum-tube technology, they would not have been economically feasible because of manufacturing and operation costs and a lack of reliability of the countless discrete components required to build them [19].

The key invention that eliminated this bottleneck to evolution and imagination is the transistor-based integrated circuit (IC), colloquially referred to as "computer chip". The inventor of the integrated circuit, Jack St. Clair Kilby, noted at his Nobel Lecture how remarkable the evolution that followed this keystone invention was: "The reality of what people have done with integrated circuits has gone far beyond what anyone – including myself – imagined possible at the time" [19]. This reality of what people have done with integrated circuits is nicely described by Moore's law. Gordon Moore, the co-founder of Intel, forecast that from 1965 onward, the number of transistors on a leading-edge integrated circuit would at least double every two years. While Moore's law cannot continue indefinitely – it has been an accurate pacemaker of the industries' development of the last 50 years – much longer than what Moore expected [13].

The increase in the number of transistors has enabled an increase in performance, capabilities, and a significant reduction in cost – at the cost of significant complexity. In the context of System-on-Chip (SoC) development, complexity refers to the number of components, the number of interactions between those components, and the difficulty of understanding and working with the models or code describing the artifacts underlying the components. Today's SoC designs consist of billions of transistors and hundreds of heterogeneous components – embedded into a single chip. These elements are all required to enable the development of smaller, faster,

smarter, and more energy-efficient electronic devices as demanded by the market.

Seeing the growth in complexity and the progress that came with it, it is easy to intuitively infer progress from complexity. Increasing complexity and progress are however not the same thing. Complexity is a necessary evil for making progress, and the complexity that arises from this progress hinders future progress. The drawback of increasing complexity in the design task can be clearly seen in today's semiconductor landscape where the high level of complexity in integrated circuits slows innovation and progress. An analysis by Collet et al. in the 2013 McKinsey study on semiconductors analyzes the previous decade with respect to changes in design complexity and design productivity. The analysis finds that in this period, the number of transistors that can be manufactured increased by a factor of 100x, yet the productivity of design increased only by a factor of 20x. It thus shows a further widening of the *design productivity gap* by a factor of about 5x in one decade [61, 46, 4].

1.2. Management of Complexity in IC Design

To tackle the growing complexity of IC design and the cost and time-to-market challenges associated with it, the semiconductor industry has developed a wide range of approaches which can be summarized in the following categories:

Specialization An increased level of specialization is applied to maintain high productivity and to provide the short cycle times demanded by increasingly challenging time-to-market requirements. The distribution of the overall design tasks onto specialized teams which can execute the individual steps more productively and the parallel execution of design tasks by different teams has resulted in the reduction of time-to-market and the ability to handle more complex designs, without however addressing the increasing design cost and the need for synchronization of common concerns.

Tools The introduction of new languages and tools to increase efficiency and productivity throughout the development workflow. The last disruptive productivity increase in this field has been achieved by RTL synthesis tools. RTL synthesis managed to raise the level of abstraction and completely automate the construction of the lower-level implementation. Other technologies which promised to deliver disruptive improvements for the entirety of the digital design domain, for example, the introduction of Transaction-Level Modeling (TLM) or High-Level Synthesis (HLS) have not been able to deliver on their promises. It is theorized that disruptive levels of improvement were not seen with TLM as it does not manage to define one agreed-upon level of abstraction – an obvious contrast to what RTL Design (synchronous design, time discretization) and Gate-Level Design (digital design, value discretization) deliver [4]. While HLS came with an increase in productivity in a limited sub-domain of digital design, it failed to raise the abstraction of the entire domain [52]. Further noteworthy examples which bring similar benefits to limited areas of the IC development task are formal methods for verification and more powerful tools for design space exploration. These specialized tools and approaches however also increase the

complexity needed to synchronize common concerns across heterogeneous development environments.

IP-reuse The reuse of existing components and artifacts across different designs. Since the early days of the semiconductor industry, the basic building blocks of digital designs were implemented once and then reused multiple times in the same or in different designs. Although IP-reuse brought a big initial leap in productivity, its continuous benefit is often overestimated [61]. While reuse can significantly reduce the cost of designs, it limits the ability to change and innovate on existing components and often requires the design of library components that exceed what is needed for any individual device – resulting in higher design and manufacturing costs.

All three pillars are essential and their application has kept the growing productivity gap in check. Their application however comes with its own challenges. As mentioned above, a high degree of specialization and increasingly large, heterogeneous tool landscapes cause significant effort for the synchronization of single source data. IP-reuse in turn has limited the ability to adapt individual components to their concrete needs. The following section describes how code generation can both address these challenges and provide an important additional component for managing complexity.

1.3. Management of Complexity Through Generation

Code Generation is an important orthogonal component to the pillars described in Section 1.2 and addresses many of the shortcomings and problems caused by their application.

Specialization and Tools Code generation plays an important role in IC development flows which rely on a high degree of specialization and make use of heterogeneous tools for different specific domain tasks. In modern, highly distributed development flows, generation can provide artifacts for different specialized tasks across different design steps and environments used by different teams and across different domains such as hardware, software, and verification. This is especially true in an environment where different tools, provided by different vendors require the same artifacts in often non-standardized or non-standard-compliant formats. Automated generation guarantees consistency and flawless synchronization and ensures that all tool-based workflows can be performed on correct and consistent data without the need for manual adaptation.

IP-reuse Generation can overcome a key limitation of IP-reuse: it addresses the limited ability to change and innovate based on existing library components by making them more configurable and adaptable.

Ecker et al. [63] reported a productivity increase by a factor of 20 in special design tasks and up to three times higher productivity in design implementation from specification freeze to tape-out through the use of Metamodeling and code generation. The replacement of ad-hoc script-based

approaches with a more structured approach and the link to formalized specification data have been mentioned as key to this improvement.

This thesis strives to establish the vision of generation as part of a new approach to designing chips as postulated by Nicolic [67] and Shacham et al. [56]. Following this vision, generators should not only be used to make simpler and more efficient designs but also to generate different alternatives, thus enabling exhaustive architecture analysis. The use of generators [4] instead of models has also been claimed by Bachrach et al. [58].

Besides controlling and configuring generation, the development of generators is the most time-consuming effort to enable automation. To achieve the vision of generation as part of a completely new approach to designing ICs, this thesis makes heavy use of the research and achievements in the field of software engineering regarding Modeling, Metamodeling and Model Driven Architecture (MDA) and provides a transfer of these achievements into the domain of IC design. With MDA, the Object Management Group (OMG) offers a vision for target code generation from specification via a set of models and a set of transformations, each deriving a model from the adjacent model [69, 42, 4].

1.4. Outline

Chapter 2 describes how advances in development languages have increased productivity and abstracted complexity in both software and hardware development. It details the shortcomings and limitations of language-based approaches and shows how model-centric development approaches can be seen as the next step to managing complexity and raising the level of abstraction of the design process. In this context, the chapter also introduces the theory underlying the Metamodeling-based code generation that has replaced the ad-hoc script-based approaches – eventually leading to the 20x improvement in some areas of the design space [63, 4].

Based on this theoretical foundation, the thesis introduces the vision of Model Driven Architecture (MDA) as postulated by the Object Management Group (OMG). It shows how the realization of this vision in the field of Software Engineering has delivered on its promise of productivity improvement. This leads to a key observation of this thesis: the vision of Model Driven Architecture has not yet been fully realized in the domain of IC design. This realization is the basis for the envisioned approach described in Section 2.5 of this thesis: this work applies the vision of Model Driven Architecture to digital hardware design and provides a wider framework for the application of Model Driven Architecture to different environments in the domain of IC design. Based on the background of this chapter, Section 8 gives an overview of the key contributions made by this thesis.

Chapter 3 presents related work. The high-level goal of this transfer is to provide more powerful automated code generation to increase productivity and handle the growing complexity in the domain of IC development with an application to digital hardware design. The chapter

thus describes related work applying and improving code generation in the field of digital hardware design. It also identifies the shortcomings that even the most advanced of these approaches cannot overcome.

Chapters 4 and 5 describe in detail the key contributions made by this thesis.

Chapter 4 shows the transfer of the formal structure of the Model Driven Architecture Vision onto the field of IC design. This includes an adaptation of the vision and its terminology for the world of IC design and the development of models and Metamodels for the Model Driven Architecture flow for digital hardware design. It is important to note that Chapter 4 provides this entirely in the context of models and Metamodels. The transfer provided here is agnostic of any concrete hardware or software development language or the concrete implementation of any framework.

Chapter 5 then describes the Python-based implementation of the proposed MDA framework. Here, it is described how automation and code generation can be used to make the development of an MDA framework and the application of this framework for the development of new generators as easy as possible. This chapter also describes what can be seen as the *heart and soul of any model-driven development approach* [45, 26]: the Model-to-Model transformations as central components of generators built on MDA.

Chapter 6 shows how the work in this thesis has been applied and extended.

Section 6.1 shows how the methodology developed in this thesis has been applied to develop generators for digital designs. It describes the commercial development projects that have been started based on the methodology developed in this thesis and shows how it is integrated into Infineon's design flow and development at Infineon Technologies AG. Section 6.2 describes how the configurable generators can be used for design space exploration. Section 6.3 shows how the methodology has been further extended outside of the scope of digital hardware design for hardware verification and the development of embedded software.

Chapter 7 demonstrates how the application of the proposed MDA flow allows users to benefit from established software development methodology and use it to solve problems in the domain of digital design.

Chapter 8 summarizes this thesis, points out its key contributions, and lists the key publications made as part of the work on this thesis.

2. Languages, Models, Metamodeling and Model Driven Architecture (MDA)

This chapter introduces the concepts of models, Metamodels, and Model Driven Architecture and describes their role in the modern software engineering landscape. It explains how these methods have demonstrated to handle complexity, and increase productivity and why they are considered a vital component of any present and future development ecosystem by many in the software engineering community. To do this, this section first summarizes the history of software engineering and describes how modeling needs arose in this domain. Next, this chapter shows how these methods are applied in software engineering where they show benefits.

Based on this understanding of the existing software engineering methods, this section identifies approaches used in the field of IC design which resemble those of software engineering – either inspired by them or developed independently with the same objectives.

This juxtaposition of model-based software engineering and IC design methods eventually leads to the main goals of this section: It identifies the Model Driven Architecture concepts that do not yet have a parallel in the domain of IC design and describes the approach envisioned in this thesis: applying the full vision of Model Driven Architecture – a concept targeted at the development of complex enterprise software landscapes – in the domain of IC design.

2.1. Evolution of Languages and their Productivity Impact

This section presents a common taxonomy of software programming languages that emphasizes the historical evolution of these languages towards higher productivity: the grouping of languages into *Programming Language Generations*. It then shows that the steps in the evolution of digital hardware design permit a grouping into categories that are similar in their definition to the Programming Language Generations defined for software programming languages.

Based on this grouping, it identifies a shortcoming shared by both hardware and software development landscapes.

2.1.1. Software Programming Languages Generations

The history of programming computers – which is now the vast field of software development – shows a clear trajectory towards increasing efficiency and handling complexity through an

increase in abstraction.

The evolution of programming languages is a good example of this. Programming languages are the main tool for creating software and have undergone a significant evolution since the development of the first computers. A commonly used grouping is the assignment of languages to *programming language generations*.

First-generation languages (1GL), also referred to as *machine languages* and represent the most primitive form of language with the lowest level of abstraction. Programs written in first-generation languages are directly recognized by a processor. They are therefore highly machine-dependent and hard to write and understand as the numerical notations of these languages are equivalent to the instruction set supported by the processor running the program. This instruction set is optimized for the fast and easy execution of a program and not to simplify the development of programs.

Second-generation languages (2GL), also referred to as *assembly languages* are the first level of abstraction in the domain of programming languages. The structure of assembly languages still closely matches the instruction set of the underlying computer. Assembly language instructions typically map one-to-one to the instructions in the instruction set. The benefit of assembly languages is their support for symbolic names (mnemonics) for operations, addresses, registers and sections of instructions.

Third-generation languages (3GL), also referred to as *higher-order languages* are the last commonly established level of abstraction. This category is significantly wider than that of first- and second-generation languages. It contains all languages that provide a degree of abstraction of the instruction set that allows developers to write programs with little knowledge of the instruction set of the underlying computer. In addition to the use of symbolic names, they can be translated to different instruction sets and therefore run on different machine types. This field ranges from very low-level languages such as Algol and C to high-level languages such as Java, C# or dynamic languages such as Python. Nowadays, third-generation languages have displaced first- and second-generation languages in all but a few highly specialized corner cases. A lot of development is still happening on improving and making third-generation languages more powerful. Still, the above taxonomy defining what is first- and second-generation language and what is third-generation language or higher is widely accepted and established.

Fourth- and fifth-generation languages (4GL and 5GL) describe abstractions above of what currently exists in the category of third-generation languages. The definition of the groups of fourth- and fifth-generation languages is more ambiguous to date. Some sources place languages such as Python into the 4GL bin and differentiate from the 3GL category on attributes such as their ease of use and similarity to human language [82]. Others define fourth-generation languages as domain-specific languages [28] or define the category based on the ability of languages to make programming accessible to non-programmers (using features such as a graphical query generator, spreadsheet capabilities, and graphical screen definition) [54].

The same applies to the 5GL category which is sometimes described as the category of declarative programming (i.e. that of constraint-based programming) [82], while in other cases being equated with the field of visual development and no-code and low-code approaches [54].

In the context of this thesis, the groups of 1GL, 2GL, and 3GL are treated as closed and defined groups where 3GL includes all high-level multi-paradigm programming languages. [54, 82]

2.1.2. Mapping of Programming Language Generations to Digital Design

The terminology of *Generations of Programming Languages* is one that has been established exclusively for the field of software engineering. The characteristics of this taxonomy can however also be used to evaluate the developments in the history of digital design.

First-generation Digital Design can be matched to transistor-level, the lowest level of abstraction in digital design. On this level, individual transistors and specialized transistor structures are used to build higher-level functions such as e.g. multiplexers or logic gates. Development on transistor-level provides the lowest level of abstraction, yet allows for the highest level of optimization. Using specialized transistor circuit structures such as transmission gates, it is for example possible to develop a one-bit multiplexer with only six transistors [11]. The formalized, Fortran-based input format used to program initial versions of the SPICE simulator is an example of the languages used in the context of first-generation digital design [7].

Second-generation Digital Design can be matched to gate level, the next higher level of abstraction in digital design. On this level, functionality is not compiled from individual transistors but from pre-assembled building blocks. The level of optimization possible on this level is already reduced - compiling a multiplexer from library logic gates would require at least 14 transistors. Similar to the second-generation software languages, assembling a design using gate-level design requires the engineer to himself pick the individual library blocks and connect them together – this requires an awareness of the exact combinational logic that is instantiated in the design [11]. The Electronic Design Interchange Format (EDIF) or a subset of the Verilog language that corresponds to the capabilities of EDIF are languages that are used to describe artifacts of second-generation digital design [8].

Third-generation Digital Design can be matched register-transfer level (RTL). RTL design is today's standard approach to the design of digital circuits. The standard languages that are nowadays used for digital design are on the level of abstraction of RTL. In RTL design, the level of abstraction is raised to registers and the definition of the logic function between registers. RTL synthesis tools handle the translation from logic and register definition onto the gate-level representation of a design. The established languages in this area are VHDL, Verilog, and SystemVerilog, with SystemVerilog developing into the defacto industry standard and VHDL and Verilog-only designs being slowly phased out. For simplicity,

this thesis treats Verilog as a subset and predecessor of SystemVerilog. Similar to the development of software languages belonging to the category of third-generation language, the languages in this field are still undergoing continuous evolutionary development – admittedly at a much slower pace than the developments in the field of third-generation software languages [11].

Fourth-generation Digital Design can be matched to all design approaches above traditional RTL design. The most well-known candidate for such an approach is high-level synthesis (HLS). The languages used to describe high-level synthesizable hardware models are typically a subset of C or C++ and SystemC. The supported libraries and language subsets are highly vendor-dependent and there is no commonly agreed definition of a high-level synthesizable language subset. When comparing high-level or behavioral synthesis approaches to levels of abstraction in software development, a resemblance between imperative and declarative programming on the software side and RTL design and high-level synthesis on the hardware side becomes apparent. In both domains, feasible and helpful approaches exist – yet their commercial applicability is currently limited to very narrow fields. For high-level synthesis, a clear benefit of the technology is visible only for datapath-driven designs with an ability to benefit pipeline-based approaches to data processing. The first generation of HLS tools was applied for IC design in the 1980s. Despite this 40-year-long history and significant development steps in the technology, it is not foreseeable that HLS becomes the next common level of abstraction that might replace the register transfer level. It is therefore also not clear whether HLS is what will fill the role of *fourth-generation* digital design [52]. The Bluespec languages (Bluespec System Verilog and Bluespec Haskell) can further be considered as fourth-generation digital design languages in the context of this thesis. The definition of state as registers and memories is clearly still a part of the Bluespec concept, it therefore still meets an important characteristic of RTL design and differentiates itself from HLS. Moreover, the Bluespec approach relies on describing the change to those stateful elements on RTL level. What Bluespec however does not need is the definition of actual RTL connectivity or scheduling. It is the compiler of the Bluespec language that takes care of all synchronization, the generation of state machines, and synchronization. This is a significant level of abstraction over the RTL level, leading to the categorization performed here [55, 30, 79].

2.1.3. Common Observations about Software and Hardware Languages

Grouping approaches to Digital Design into the same categories that are used for software development languages allows us to come up with important common observations about both fields:

1. For both hardware and software development, there is noteworthy ongoing development in the category of third-generation languages. These new third-generation languages have drastically reduced the complexity of development tasks and allowed developers to tackle problems of a complexity that was undreamed of in the days of first-generation

programming languages and initial hardware description languages [21].

2. All these new developments of efficient third-generation languages are however an evolutionary approach that has a known upper boundary: In both hardware and software domains, the law of diminishing returns manifests: ever more complex and powerful languages (which are therefore harder to learn) provide ever smaller returns in developer productivity. Examples in the domain of software engineering are the increased complexity of the C++ language and the sheer amount of different languages with relatively small developer communities. In the domain of IC design, the complexity of SystemVerilog is an example of the limits of productivity which can be obtained by improvements in third-generation languages.

3. The abstraction the third-generation languages provide is sufficient for simple software and smaller IC design projects. For the complexity of large, state-of-the-art products in both IC design and software development, it is clear that both the current third-generation languages and the discernible upper threshold of productivity a potential future third-generation language can provide are insufficient [23]. One of their key issues is their inability to provide human and machine-readable views of complex systems, their complex structure, and their complex interdependencies. In neither of the domains, a clear path to a *fourth-generation* approach can be identified and the replacement of third-generation languages is not anticipated.

We can derive a growing productivity gap from these observations: improvements in third-generation languages increase productivity at a decelerating rate while the complexity of designs is increasing at an accelerating rate. The following sections show how model-based, language-independent approaches can contribute to reducing this gap.

2.2. Model-based and Language-independent Approaches

The ambiguity regarding language categories above *third-generation languages* highlights the need for and the importance of model-based approaches. Model-based approaches are able to address the shortcomings of third-generation languages described in the previous section and can be seen as language-independent continuation of the raising levels of abstraction the language generations provide.

This section details modern model-based development approaches and demonstrates how they already provide the first answers to the question of what comes above the third-generation languages. To do this, it first broadly categorizes the levels of modeling which are applied to industry development tasks into a spectrum of modeling. It then introduces Metamodeling as the state-of-the-art for efficient modeling and code generation. Based on this theoretical underpinning, it shows how the technology is used as part of software and hardware development and details its benefits and weaknesses.

2.2.1. The Modeling Spectrum

The application of modern model-based approaches differs depending on the domain and complexity of the problem at hand. This section presents a view of the modeling spectrum as defined by Kleppe et. al. [23]. Figure 2.1 describes the different categories of modeling used.

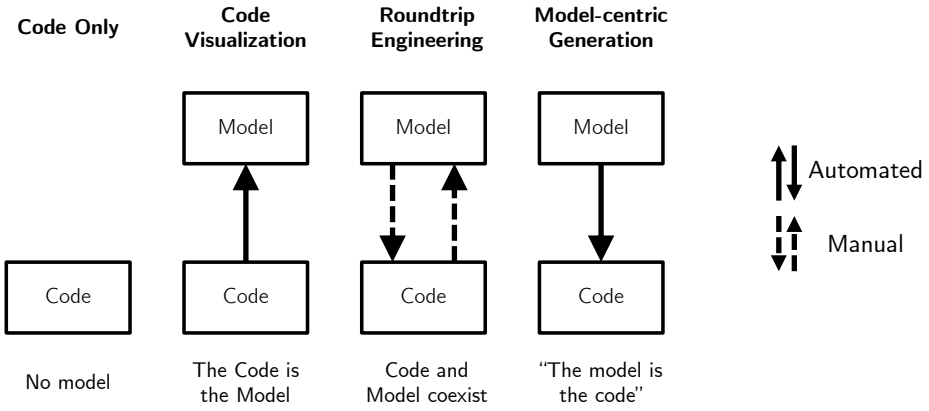

Figure 2.1.: The Modeling Spectrum [23]

Code Only development does not utilize any modeling and relies only on code as the primary artifact to define a software or hardware product. Any abstractions of the code only exist in the developer's mental model of the implementation.

Code Visualization approaches use models to reduce one of the main weaknesses of Code Only approaches: it extracts modeling artifacts from the code that can be used by humans to get a better view of the structure and interdependencies inside these systems. In this scenario, the code acts as a model and it is only possible to visualize model artifacts that are captured in the code itself. Higher-level modeling artifacts like ownership of objects, which may not have a counterpart in the language used to implement the code cannot be captured in code and therefore not be visualized. The extraction of models from the code is commonly done automatically which ensures consistency and reduces effort.

Roundtrip Engineering describes an approach where a formal, code-independent model is maintained by the engineering team. This model is manually kept in sync with the code that it describes. Any changes to the model that impact the code require corresponding changes in the code and vice-versa.

The main benefit of this approach is that the model can contain additional modeling artifacts that need not be supported by the language of the code. Because of these additional artifacts, it is however no longer possible to automate the generation of the model from the code.

Moreover, while many approaches support initial code generation from the model, the generated code artifacts are intentionally loosely coupled to the model. It is intended that they are extended and modified by the developers. As a result of this intentionally loose coupling, an automated change to the code after a change in the model is not possible in the practical incarnations of this approach.

Roundtrip engineering helps with the understanding of the complexity in the underlying code and provides a better abstraction than Code Visualization – it however does not remove the complexity from the development task. This can be compared to how assembly language behaves towards machine code: assembly language abstracts addresses and bit codes and therefore simplifies writing and understanding code – it however does not remove the burden to understand the intricacies of the instruction set of the underlying machine.

Model-centric Generation is the first automated, fully model-based generation approach. When this approach is applied, a certain aspect of a software or hardware system is fully described by a model. From this model, a fully automated generator will provide the code.

Similar to Roundtrip Engineering, this approach allows models to contain additional artifacts that may not be supported by a particular generated code view. This enables the scenario depicted in Figure 2.2: a model can contain the superset of all artifacts required to generate target views of different domains (e.g. code of different languages, for hardware development and software development, for software or hardware verification activities, hardware validation or documentation).

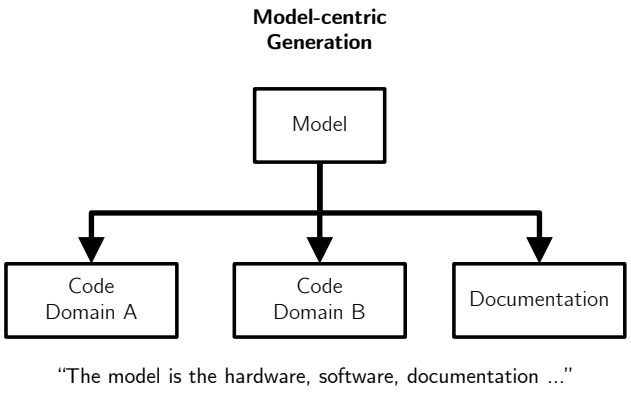

Figure 2.2.: Model-centric generation of multiple outputs

The most noteworthy difference to Roundtrip Engineering is the fact that – when the philosophy of Model-centric Generation is followed – any generated artifact must not be manually modified. As a consequence, for any change in the model, a new target view can automatically be generated.

Model-centric generation can take the next step to both ease the understanding and analysis of

complex systems and remove complexity from the development task. The development teams can rely on the generator and no longer need to carry the burden of understanding the intricacies of the generated code. This leaves the development teams with a view that is significantly more abstract and less complex than what can be achieved with code developed in a third-generation programming language.

It is important to note that modern software and hardware development projects do not just rely on one of the approaches described above. For any reasonably large, heterogeneous development project, a combination of all approaches from the Modeling Spectrum is applied. Productivity benefits can be obtained by moving more and more development tasks toward Model-centric generation. This can be achieved by either identifying coding tasks that can be solved with existing Model-centric Generation methodologies or by developing new generation methodologies.

The state-of-the-art of existing Model-centric generation methodologies is described and their existing applications are described in Section 2.3. Model Driven Architecture as a more advanced and significantly more powerful methodology with a wider field of application is described in Section 2.4.

2.3. Metamodeling and Metamodeling-based Code Generation

Based on the overview of model-centric development provided in the previous section, this section will detail the state-of-the-art of Metamodeling, a structured and clean approach to model-centric automation.

2.3.1. Simple Code Generation without Metamodeling

Developing generators that generate views from input models can be achieved with simple scripting approaches – a practice that is common in both the software and hardware industry [16]. In the following, these scripts are referred to as point-solution generators.

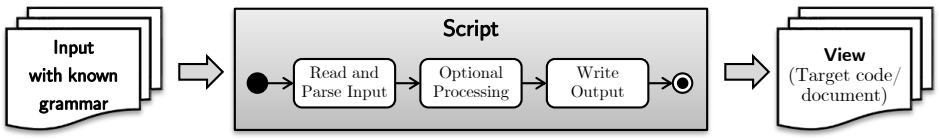

Figure 2.3.: Illustration of a simple, unstructured script-based generator

The only prerequisite for a point solution script is structured input data that follows an explicitly or implicitly defined grammar. Figure 2.3 shows how such a script typically generates its outputs: it reads a known input file and parses its content, applies some optional, simple processing on the data, and then uses either file I/O or standard output to write the view information.

This simple approach lacks many important requirements for a structured, re-usable and extendable model-centric automation framework:

1. A language and data format independent definition of a model's structure.
2. An automated and reusable approach to checking correct model structure.
3. An automated and reusable approach to serializing and deserializing models for storage.
4. A method to facilitate the visualization and entry of models and model structure.
5. A re-usable and structured approach to define mappings from models to views.

These requirements are met by state-of-the-art Metamodeling as detailed in the rest of this chapter [21].

2.3.2. Metamodeling Concept

Metamodeling is an important theoretical foundation for a structured approach to code generation. The prefix "meta" can be loosely translated to "after" or "beyond". In the term Metamodeling, it is used to highlight that a Metamodel is a *model of a model*. A Metamodel is an abstraction of a set of models, defining the model's properties and specifying the relationships between its elements [60, 53].

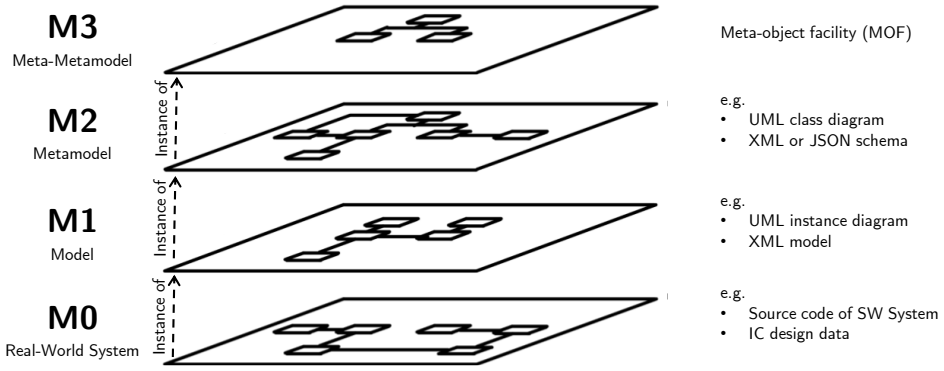

Figure 2.4.: Metamodeling, Hierarchies of Models as defined by OMG [21]

The relationship between a model and its Metamodel does not only exist on a single level: A Metamodel can in turn be conceptualized as a model, which then will have its own Metamodel. The Object Management Group (OMG) is a standardization consortium that has developed a standard formalization of these *model of a model* hierarchies. Figure 2.4 illustrates these hierarchies using the OMG standardized terminologies. For each of the layers in the figure, the model in the lower layer is in an *instance of* the model in the layer directly above it. The following will describe the characteristics of the individual layers from bottom to top.

M0 contains the *Real-World System* subject to the Metamodeling based automation. Elements on this layer are for example the source code of a software tool or library or design artifacts such as hardware description language or schematic data in the IC design process.

M1 holds the *Model* of the real-world system subject to automation. This layer is what is typically read or generated by a framework that generates views for use in the design process. The input data handled by the typical users of automation tools resides on this layer.

M2 holds the *Metamodel* of the M1 model. This layer defines the M1 model's properties and specifies the relationships between its elements. It is the layer that has to be handled by the developers of automation tools.

M3 holds the *Meta-Metamodel*, referred to as *Meta Object Facility* in OMG terminology. Any M2 Metamodel can be described as an instance of the Meta-Metamodel, however, also the Meta-Metamodel can be described as an instance of itself. This self-describing property of Meta-Metamodels is called *metacircular*. The Meta-Metamodel layer is not used by developers of automation tools but by developers of the framework for automation tools to develop tooling that can be utilized for all M2 Metamodels. These approaches are further described in Section 2.3.3.

This theoretical foundation provides a solid, language-independent view of models and the data they contain. When this theoretical foundation is used for the implementation of a Metamodeling framework, it is guaranteed that all available models are easily accessible and compatible. The theoretical foundation also permits further analysis and mathematical formalization which has been developed as part of this thesis work and is published in [3].

The next section details the practical benefits that the theoretical Metamodeling framework has for the structured development of generators.

2.3.3. Capabilities and Advantages of Metamodeling Frameworks for Code Generation

This section describes the structure and features a generic, language-independent Metamodeling framework has. The described structure and features match those of the Metamodeling framework implemented at Infineon Technologies AG and used for the work of this thesis, the descriptions are however deliberately kept generic and will apply to any metamodeling framework, regardless of the concrete implementation.

Figure 2.5 shows an overview of the components such a Metamodeling framework provides for a given Metamodel. The arrangement of the blocks visualizes the data flow through the framework from left to right. The white boxes describe the input or output of the metamodeling framework. Possible inputs are either an M1-layer model used to generate the views (boxes "Specification" or "Persistent Storage") or an M2-layer Metamodel ("Metamodel Description")

used to generate and configure the framework itself. Possible outputs are in turn M1-layer models "Persistent Storage" or generated target views such as software code, hardware description code, or documents ("View" or M0-layer)). The elements depicted in solid black ("Deserialization", "API and Validation" and "Serialization") are components that are typically provided by the framework out of the box, and the elements depicted in light gray ("Reader" and "Generator") are elements that need to be manually developed as extensions to the framework. They provide the ability to read data from data formats custom to a certain M2-layer Metamodel or to write data to such a format.

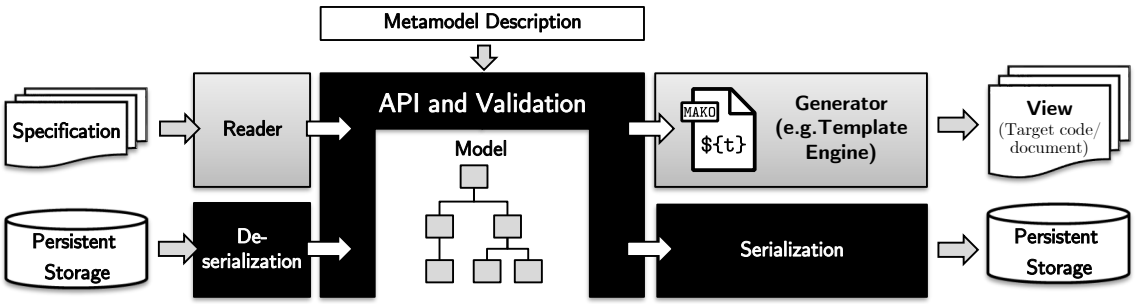

Figure 2.5.: Sample diagram of a Code Generation Framework based on Metamodeling

The Sample Framework depicted in Figure 2.5 already highlights many of the advantages of an approach based on a formalized Meta-Metamodel and a common framework.

These advantages address all the requirements identified from the shortcomings of scripting-based generators in Section 2.3.1:

1. A definition of a model's structure that is independent of language and data format: The Metamodel Description itself is just a model. It is therefore possible to store this model persistently along with other models, to process it as part of the modeling framework, and to use it as input for generators.

2. An automated and re-usable approach to checking correct model structure: Based on the Metamodel, the Metamodeling Framework can generate data structures, Application Programming Interfaces (APIs) to wrap them and Data Validation primitives to ensure that any generated models adhere to the Metamodel.

3. An automated and re-usable approach to serializing and deserializing models for storage: Based on the Metamodel, the Metamodeling Framework can generate code and tooling to serialize and deserialize models into different formats. This can include text-based formats such as JSON and XML, binary formats such as Protobuf or Flatbuffers, as well as Interfaces to Databases.

4. A method to facilitate the visualization and entry of models and model structure: Based on the Metamodel, the Metamodeling Framework can generate or customize graphical user interfaces for model data entry.

5. A re-usable and structured approach to define mappings from models to views: As illustrated by Figure 2.5, all generation tasks can be split into reading or deserializing the data from a certain input format into the generated API and the serialization or writing of either the model or a generated view.

   This structure permits reuse from two sides: reuse on the reader side is possible as multiple generators can use the same readers and APIs for different views. Reuse on the Generator side is possible as different inputs (e.g. different Specification formats) can be mapped onto the same API and utilize existing generators for this API.

   Point-solution generators as described in Section 2.3.1 which are not built on a Metamodeling framework do not typically provide a clean separation between these components and cannot enable the same degree of reuse.

In addition to addressing all requirements, Metamodeling frameworks have several further benefits

- Meta-Metamodels are defined in a language-independent manner. It is therefore possible to provide Metamodeling frameworks that bridge language boundaries and allow different generators or readers to be implemented in different languages. This provides a high degree of interoperability between heterogeneous systems.

- Based on the Meta-Metamodels, it is possible to generate further infrastructure for all Metamodels. Any such extension of the Metamodeling framework can benefit all use cases of the framework, regardless of the Metamodel on which they rely.

- The structure of generators and their development is simple and straightforward. This even permits the implementation of generators by domain experts and reduces the overhead of communicating specialized domain knowledge between tool developers and domain experts.

2.3.4. Application in Software Engineering

In the software domain, generator-based approaches are particularly successful when it comes to interfaces between systems. This makes the methodology important for heterogeneous systems spanning different languages and frameworks.

Machine-readable interface definition languages (IDL) such as OpenAPI are an important pillar of modern software development, both for distributed microservice architectures and for traditional monolithic client-server applications. Common software development tooling uses interface definitions to automatically generate documentation, interactive API browsers, and clients utilized mainly for debugging purposes as well as the actual server or client implementations of the defined interfaces. These interfaces include the data structures that are defined as part of the IDL, as well as the serialization and deserialization needed for the formats such as

GraphQL, gRPC, or REST calls with JSON and XML payload. These approaches contribute an important piece to the puzzle of modern application development: they manage to completely hide the complexity of communicating between front- and backends implemented in different languages and allow seamless exchange of data between microservices [84, 51, 87, 73].

A strong piece of evidence that model-based generation is an integral part of software development environments is presented by recent developments in the Roslyn compiler framework. This compiler framework includes code generation in the compiler. As part of the compilation process, the Roslyn compiler framework can invoke custom code generators which can be added by developers through the dependency management of their project. These code generators can then use input models and emit source code into the ongoing compilation. This is not limited to existing input models with known Metamodels which can be manually provided to the compiler: it is possible to analyze the Abstract Syntax Tree (AST) of existing source code, build custom models from it, and inject code into the compilation that extends it. This process is basically the equivalent of introspection at compile-time [94, 96].

2.3.5. Application in IC Design

Considering the fact that model-centric code generation in software engineering is particularly helpful when heterogeneous systems need to be connected, it does not come as a surprise to see that model-centric code generation is even more successful in the domain of IC design: the heterogeneity of this domain is one of its key characteristics.

Metamodeling-based Code Generation has demonstrated its ability to bridge this heterogeneity. So-called *Single Source Flows* based on or derived from the industry standard IP-XACT is indispensable to modern IC design [47]. IP-XACT is an XML data format with a well-defined underlying Metamodel used to describe cross-functional aspects of all IP[1] components (reused or custom designed for a certain IC) [36].

As part of a single source flow, the memory and register addresses described by the IP-XACT representation of the individual components are combined with information on component instantiation and system-level address maps. This combined information can be used to generate diverse output artifacts (e.g. customer documentation, C files for software development or test engineering, and even hardware description language view defining the registers and their address map inside the IPs). These tools eliminate the tedious and error-prone manual work to keep these artifacts consistent between different areas. They also eliminate the risk of costly mistakes introduced during the manual transfer of data between the domains. Eventually, they simplify re-work and reduce the time and cost penalty of changes as artifacts that are automatically generated from single sources can easily be regenerated.

This makes it easy to understand the productivity increases by a factor of 20 in special design tasks and up to a factor of three in design implementation from specification freeze to tape-out

[1] The term IP (intellectual property) describes re-usable electronic circuit design components

through the use of Metamodeling and code generation [63]. While single-source generation approaches have become increasingly widespread and successful, they are not a panacea for productivity increase. Their field of application is so far limited to relatively simple configurability and parameterization. The reuse of existing components is therefore still limited by the static nature of the re-usable building blocks (IPs). These limitations are further analyzed in the following section.

2.3.6. Limitations of Model-based Generation Approaches

Generators are more useful the more manual work they automate and the more complex the automated work is. Figure 2.6 illustrates the challenge that raises towards generator design: the input data or specifications fed into the generator should be at the highest possible level of abstraction while the generated artifacts need to be at the low level of abstraction needed by the output tools and workflows consuming them.

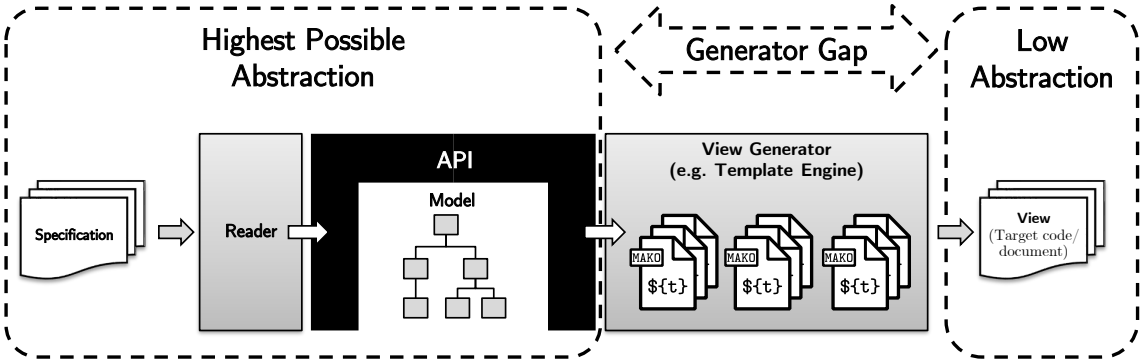

Figure 2.6.: The Generator Gap in Simple Metamodeling Approaches

The push to make generators more ubiquitous and powerful leads to ever higher levels of abstraction on the input side – with an ideal target of full-chip generation from high-level specifications. Moreover, the level of abstraction of the target views is reduced by the need to generate views that consider more and more low-level physical properties of the targeted hardware, such as timing budgets or low-power mechanisms. In consequence, the complexity that generators need to bridge is significantly increasing, making the development of generators difficult and costly.

In this work, this increasing gap between specifications and low-level target views is referred to as *Generator Gap*. Powerful code generation approaches should bridge a Generator Gap that is as large as possible, i.e. provide the ability to generate target views of a low level of abstraction from specification inputs at the highest possible level of abstraction. The complexity of the generator development is the main hurdle to increasing the level of abstraction of specification inputs for any target view. Bridging this gap, for example by identifying novel ways to make the development of code generators easier, is important both to increase the level of abstraction for inputs of existing generators as well as to find new fields where model-centric automation can

be applied. The following paragraphs elaborate on the sources of the complexity that makes the generator gap difficult to bridge:

1. A main driver of generator complexity is the mix of business logic used to analyze, understand, and automatically refine the input model (i.e. specification) to the level of target view with the logic that emits the language or format of the target view. For example, conditional statements of the generator logic and conditional statements emitted into the target view are inevitably intertwined in the generator code. Separating these aspects becomes particularly difficult when the language of the generated target view is similar or identical to that of the generator.

2. Another driver of generator complexity is the amount of configurability that is needed for the generated views. When generators are used across different projects in different domains and departments, the need for different generator flavors inevitably arises. This configurability increases the complexity of the generator code as it has to be considered in addition to the business logic responsible for automatically refining the specification. In many scenarios, the number of configuration flags required for an individual generator massively impacts the code size and complexity of the generator.

3. Generators are closely intertwined with the target platform they were designed for. This causes two issues:

    a) Generators use the same specification and Metamodel to generate different target views. The different target views typically have many common aspects. The higher the level of abstraction of the generator input, the more common business logic is needed to automatically refine the input specification to the level of abstraction of the target views. In an industrial context, it can be observed countless times that development teams fail to centralize these break-down efforts and therefore implement duplicate business logic in multiple generators to achieve the breakdown. This increases the overall complexity of the generator development task and causes subtle, difficult-to-spot inconsistencies between different generators targeting similar outputs. A structured approach to centralize these efforts is needed and the model-centric generation approaches described so far have failed to reliably deliver them.

    b) It is difficult to add new generators or modify existing ones for a new target platform. This can be achieved by adding further outputs to existing generators. Doing so is challenging as it requires an understanding of the existing generators and their current outputs. Problems with this approach are very similar to the issues sketched in point 2. Alternatively, a new generator can be developed. It has been observed that development teams fail to extract commonalities with existing generators into library elements without breaking existing generators. This alternative approach thus results in duplicate business logic in multiple generators.

The complexity and issues described above are significant hurdles to further progress and prohibit extending the use of model-centric code generation across the software and hardware

development landscape. A systematic, structured, and reusable approach is required to resolve these issues. The following section introduces Model Driven Architecture and describe its promises to address these sources of complexity and to open up the path to further automation.

2.4. Model Driven Architecture (MDA)

Model Driven Architecture (MDA) is an approach toward model-centric development that was postulated by the Object Management Group (OMG), an industry consortium with a focus on providing standards for interoperability in software development. The group's initiative to standardize MDA was launched in 2001 and has significantly influenced the principles and practice of model-centric software development.

Model Driven Architecture relies on the fundamental cornerstones of model-centric development, Metamodeling, and Metamodeling based Code Generation as introduced in the previous sections. On these cornerstones, Model Driven Architecture postulates the vision of code generation via a chain of models and a series of transformations between models – together with an architectural framework of layers in which these transformations are grouped.

The following sections introduce the MDA idea in detail and describe its application and benefits for software engineering [23, 42, 69, 25].

2.4.1. The Model Driven Architecture Vision

In a development flow based on Model Driven Architecture, models are the primary artifacts throughout the entire development life-cycle [34]. Metamodeling is an integral part of MDA. It is relied on to describe models in a formalized way and thus permits automated generation from models as well as transformations between models. In Model Driven Architecture, Metamodels are used for all models, starting from a very abstract specification level down to the concrete implementation levels. The formalization Metamodeling provides allows to automatically transform and refine models into more fine-grained models which are eventually used to implement or automatically generate the intended targets.

2.4.1.1. CIM, PIM, and PSM Models and Metamodels

Figure 2.7 illustrates the role models and Metamodels play in Model Driven Architecture. The first version of MDA introduces three kinds of models, namely a Computation Independent Model (CIM), Platform Independent Model (PIM) and Platform Specific Model (PSM). The models on each of these layers are formalized by a Metamodel (see the "instance of" relationship depicted in the figure).

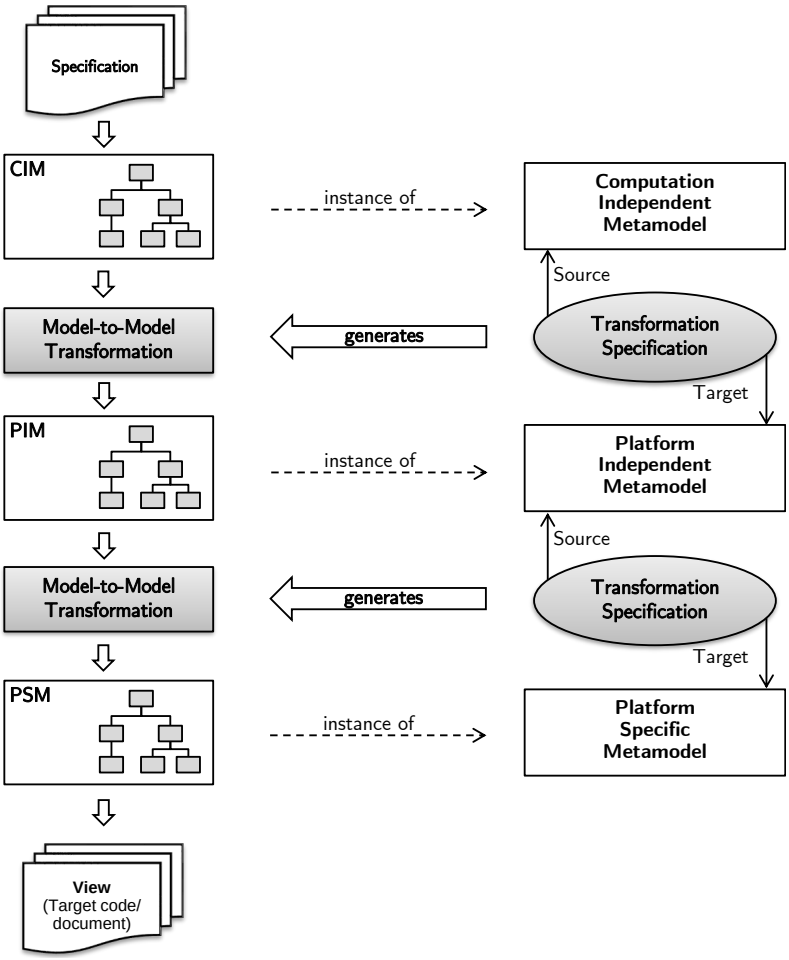

Figure 2.7.: Model Driven Architecture with its Model Transformations and Transformation Hierarchies as defined by OMG [23]

CIM The Computation Independent Model is the most abstract one and closest to specification. It considers neither detailed algorithm implementation nor architecture. Computation Independent Model is a legacy term from the first and most abstract specification of Model Driven Architecture which is still commonly used and also utilized in this thesis. It was initially defined as a model describing "business concepts whereas a PIM may define a high-level systems architecture to meet business needs" [25]. In the context of this thesis, the term CIM is used to refer to a model that is (relative to a certain platform) more abstract than the PIM. The CIM is always closer to the specification and less dependent on actual implementation details than the PIM.

PIM The Platform Independent Model (PIM) defines the architecture, functionality, and behavior without specific details of the target platform. In the context of MDA, something is considered a platform if it provides an environment to execute or process models or if there is an automated transformation to lower-level platforms. In the software world, a platform is therefore a computing infrastructure on which generated target code can

run. A platform for example defines libraries, APIs, and the OS the generated view code compiles and executes on.[2][3, 25, 68].

PSM The Platform Specific Model is platform dependent and closest to the target code. From this model, the view is generated. It contains information on how exactly a computation is performed on a given target platform (it refers to concrete libraries that provide required functionalities) [25].

2.4.1.2. Model-to-Model Transformations and Transformation Specifications

In a generation flow based on Model Driven Architecture, the most abstract model (CIM) is filled by reading some form of data source (referred to as Specification in Figure 2.7). The lower layers of models (PIM and PSM in the OMG terminology) are filled using the data in the model above the respective step. Formally, the refinement step that utilizes the information from the upper-level model to fill the lower-level model is a Model-to-Model transformation. The challenges that are addressed in the Generator Gap depicted in Figure 2.6 are distributed between these Model-to-Model Transformations and the View Generation.

The Model-to-Model Transformations are supported by Transformation Specifications. They describe how to transform models complying with the more abstract Metamodel into models of the less abstract Metamodel. The capabilities of these specifications can range from configuring to automatically generating the entire transformation. They allow us to customize and control the Model-to-Model Transformations [41, 34].

In many publications, the importance of configurable and generic Model-to-Model Transformations is highlighted by using a Y-Chart form (see e.g. [103]) to visualize the Transformation from PIM to PSM. Figure 2.8 contains a Y-chart visualization of MDA models and transformations. Here, a PIM (more abstract model) at the left top of the Y is transformed into a PSM (less abstract model) at the bottom of the Y using a Transformation Specification at the top right of the Y: a Platform Model (PM), which contains the details of the target platform.

The visualization in Figure 2.8 highlights an issue related to the position of the Platform Model in the Y-Chart. The Platform Model is a description of the platform a PIM is mapped onto and its level of abstraction is comparable to that of the PSM. It is therefore depicted too far up in the Y-Chart representation as it is closer to the PSM's level of abstraction.

In the context of this thesis, the Platform Model is referred to as Transformation Specification and displayed on a level between CIM and PIM (the visualization used in Figure 2.7). This is also more in line with the second version of MDA released in mid-2014 [69]. Here, every Model-to-Model transformation is considered a transformation that translates a PIM to a PSM. A sequence of any number of transformations is chained together to finally generate the view. At

[2] Please note that the term "platform" is used in different ways in the hardware design area: Keutzer et al. defined a hardware platform as a family of microarchitectures that allow sustainable reuse of SW in [39]. In other contexts, the term platform is used for a product family which addresses one field of application in various configurations.

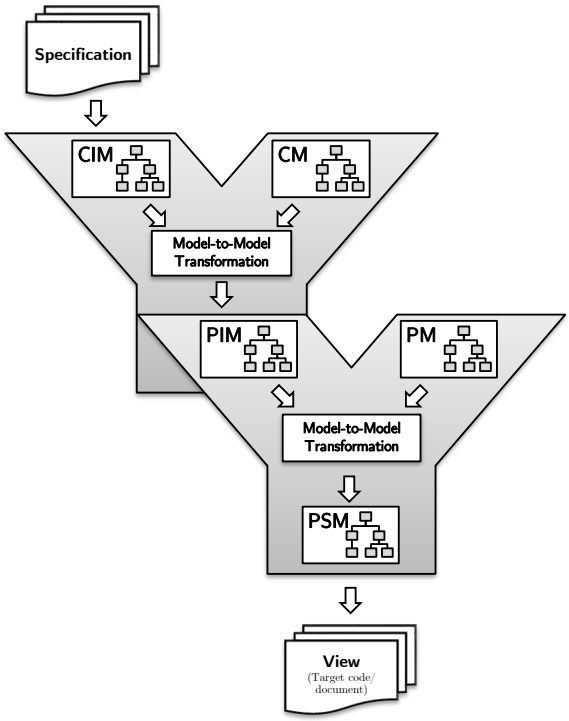

Figure 2.8.: Y-Chart Visualization of Model Driven Architecture

each connection point, the PSM becomes the PIM of the next transformation. A platform is therefore no longer simply related to a computation platform. Instead, for each platform, an automated construction path – potentially via several intermediate steps – to the target code is provided. It is important to note that the terms *platform independent* and *platform dependent* are only valid relative to a certain platform. A platform itself can in turn build on one or more other platforms [20]. The OMG illustrates this with the following example: "XML may be considered a platform but XML is independent of the language used, so it may be mapped to a language like Java or C. C is independent of the processor used so C may be mapped to a Pentium or ARM CPU." [25]

2.4.2. Application in Software Engineering

The Model Driven Architecture approach gained a high level of popularity after the initial vision was formalized by the OMG. This became visible through an increasing degree of publications and the amount of tool support for Model Driven Architecture. At the end of 2003, "MDA compliance" and the terminology of CIM, PIM and PSM were widely used for tools in the model-based software development ecosystem. More than 40 vendors used the terminology to describe their products. In the code generation ecosystem, tools like IBM Rational XDE, Codagen Architect and ioSoftware ArcStyler stood out [27, 66].

An interesting example of the success of the MDA methodology is provided by ioSoftware and its business partners. In a project in the automotive industry, the companies describe that Model Driven Architecture optimizes multi-site software development processes, provided a 15% increase in developer productivity in the first development year, and a total productivity increase of up to 30% in the second year (compared to a development process that is not based on Model Driven Architecture). The environment where these benefits were achieved stands out through its complex business logic and processes and an existing, Java-based ecosystem that integrated with external systems via HTTP and XML-based communication and Message Queues [33].

The popularity of the MDA terminology and approach has since then rapidly faded. This can be compared to OMG's CORBA standard, which rose to popularity in the early 1990s and has since then fallen in popularity and developed into an obscure niche technology that is no longer applied in modern software development [37]. On the surface, it may appear as if Model Driven Architecture has fallen victim to the same fate. It is correct that many of the tools are no longer used and maintained (the three formerly most popular commercial MDA generation tools IBM Rational XDE, Codagen Architect, and ioSoftware ArcStyler have been discontinued) and companies such as ioSoftware that contributed to implementing the MDA vision have since disappeared.

It is very important to note however that Model Driven Architecture is a vision coming together with a set of terminology, formalism, and standards. While the terminology, formalism, and standards of Model Driven Architecture have disappeared or faded into relative obscurity, the underlying vision of Model Driven Architecture is alive and well and arguably more commercially successful than ever. The following describes how the intellectual heritage of Model Driven Architecture is visible in the sprawling field of Low Code Application Development (LCAD) tools.

Low Code Application Development Platforms are software frameworks that allow developers and business experts to build and deploy applications easily with minimal coding on a level of abstraction that can be considered computation-independent and platform-independent. These platforms typically provide a visual interface on which their users can specify Computation Independent Models and Platform Independent Models of the problem domain, describing the Domain Model underlying the problem and modeling the intended control and data flow of the application graphically. Low Code Application Development Platforms significantly simplify the development process, opening the domain of application development up to non-technical users without the need for detailed knowledge of the underlying platform [83].

Low-code platforms can be utilized to develop and generate applications for different platforms including web and mobile where they are mainly suited to build data entry- and process-driven applications (for example for business process automation). They have demonstrated the ability to meet domain-specific business needs at a significantly lower cost than the traditional platform-specific development process (i.e. less time and resources needed to develop and deploy the applications). The iterative and agile development process these platforms support has proven to improve collaboration between developers and business experts - ultimately supporting the fast adoption of changing business needs. All these are aspects where the state-

of-the-art Low Code Application Development Platforms deliver on promises made by Model Driven Architecture for the well-defined niche of applications they are suitable for [86].

There is one aspect that is even more important: individual companies in the field have demonstrated that they can deliver a truly Platform Independent approach. Outsystems, one of the commercial leaders in the field, has for example migrated from a Java-based environment to a C# based one. This migration demonstrated that it is feasible to generate different Platform Specific Models from the Platform Independent Models generated by users of their tool.

Low-code application development platforms are increasingly popular in recent years and can now be considered a key component of the software development landscape. The Gartner Magic Quadrant predicts that "by 2024, three-quarters of large enterprises will be using at least four low-code development tools for both IT application development and citizen development initiatives" [78] and that "low-code application development will be responsible for more than 65% of application development activity by then" [78].

While it is possible to conclude from their capabilities that Low Code Application Development Platforms implement the vision of Model Driven Architecture, none of the commercially successful approaches provide the same clear set of terminology, formalism, and open standards that the OMG set out to establish for Model Driven Architecture. Instead, their closed and non-standard approaches do not allow for the same future proofing and platform independence that MDA postulates. This work still considers these Low-Code Application Platforms as incarnations of the Model Driven Architecture vision. Their commercial success has delivered on or even exceeded the promise of Model Driven Architecture. It is therefore strong evidence for the ability of the Model Driven Architecture vision to deliver on its promise [78].

2.4.3. Application in IC Design

Outside of the Metamodeling research group at Infineon Technologies AG, we were able to identify a few very limited attempts to apply Model Driven Architecture to IC design. There are approaches where point problems in the IC Design process are solved with approaches compatible with the vision of Model Driven Architecture. Coyle et. al. for example introduce an approach where UML state diagrams as Platform Independent Model are translated into HDL-specific views. This translation happens however directly from the Platform Independent Model, without going through a Platform Specific abstraction of the generated code [32].

As of our knowledge, there is no hardware equivalent that provides an end-to-end, top-to-bottom chip generation framework following the vision of Model Driven Architecture. We are not aware of any commercial solutions or holistic academic approaches. In particular, there is no proof of viability or commercial success similar to what is detailed for software in Section 2.4.2.

2.5. Thesis Goal and Envisioned Approach

The previous sections have demonstrated that Metamodeling and Metamodeling-based Code Generation are successful methods to deliver automation that handles complexity and increases productivity both in the domains of software development and even more so in the heterogeneous landscape of IC development. It has also identified the main challenge prohibiting the development of more powerful code generators: the *Generator Gap* described in Section 2.3.6.

The high-level goal of this thesis is to provide more powerful automated code generation to increase productivity and handle the growing complexity in the domain of IC development with a focus on the automated generation of digital RTL designs. In other words, this thesis strives to identify methods to overcome and manage the increasing *Generator Gap* that is making the development of digital design generators challenging. For this purpose, this thesis shall *identify and apply the methodology to reduce or better handle the complexity inherent to building generators.*

The methodology provided in this thesis shall meet the following requirements:

- The methodology shall reduce the complexity and cost of developing code generators for digital hardware designs. It shall do this by automating a significant part of the development tasks required for generator development and by enabling better reuse of generators.

- The methodology shall make design generators more powerful by providing the ability to raise the level of abstraction of the models used as input to the generators.

- The methodology shall extend the use of code generation into areas of digital design that are currently difficult or impossible to cover by it. It shall therefore provide the capability to generate a significant amount of the digital design artifacts that are currently manually developed.

- The methodology shall be applicable and scalable in a commercial hardware development environment. The development of generators has to be possible for hardware designers who are not trained software engineers. To enable adoption in a commercial environment, the methodology shall be easy to learn for users which are familiar with traditional code generators. Moreover, it must be scalable to the commercial needs of real-world IC products with support for complex real-world designs.

This chapter has outlined how the vision of Model Driven Architecture promises to deliver such generator-based automation. It also shows that the application of Model Driven Architecture in the software domain has delivered on its promises: there are many commercially viable products and significant productivity benefits. In particular, the developments in the rapidly growing field of low-code application development provide evidence of the disruptive power of this kind of model-based generation [78]. Similar developments cannot be observed in the field of fourth-generation programming languages: the scope of these languages is still not clearly

defined and the field has not managed to provide disruptive productivity improvements on a commercial scale.

A similar pattern exists in the domain of IC design. This work categorized digital design into RTL design (third-generation digital design) and everything beyond the abstraction level of traditional RTL design (fourth-generation digital design). While many of the methods and languages which were categorized as fourth-generation digital design are applied in a certain sub-domain of digital design and deliver a productivity increase over established methods of digital design on Register Transfer Level (RTL), they have failed to deliver the disruptive improvements and did not supersede RTL design. In the field of IC design, the methods suggested by the vision of Model Driven Architecture have not yet been extensively and systematically applied.

Our central hypothesis is that it is possible to transfer the achievements of Model Driven Architecture in the software domain to the domain of IC design and that a model-based generation approach inspired by the vision of Model Driven Architecture can deliver a disruptive improvement to the field: the approach envisioned in this thesis is to apply the vision of Model Driven Architecture to digital hardware design on Register Transfer Level (RTL).

For this purpose, the multi-layer approach suggested by the Object Management Group (OMG) for the software domain is used as a foundation for a corresponding approach in the field of digital hardware design. This thesis defines key Metamodels required for digital design and for the generation of synthesizable hardware description language views. These models shall capture the nature and semantics of the underlying domain (the semantics of digital hardware and the implementation thinking applied by hardware design engineers). The Metamodels used for the individual layers shall be designed in a way that eases Model-to-Model transformations between the layers. Based on Metamodel-based formalizations, this thesis shall define a solid and easy-to-use MDA framework and realize it on top of Infineon's proprietary Python-based Metamodeling Framework including all necessary additions to that framework to enable its use in a Model Driven Architecture context.

It is not a goal for the defined methodology to find language-based approaches that can move overall digital design methodology to a level of abstraction above that of RTL design – this is the goal that is at the center of fourth-generation digital design and thus not in the scope of this thesis. Instead, this thesis shall provide a methodology to develop powerful generator-based automation that provides automatically generated RTL views in third-generation languages from specification models that are far more abstract than the level achieved or targeted by fourth-generation languages.

The next chapter describes related work that promises to provide more powerful generators on the level of abstraction of third-generation languages. The subsequent chapters of this thesis then lay out how the concept of Model Driven Architecture has been successfully transferred and applied to the domain of IC design. This is the basis to compare and contrast the achievements of this thesis to those of the related work.

3. Related Work

Section 2.5 describes the high-level goal of this thesis: to provide more powerful automated code generation to increase productivity and to better handle the growing complexity in the domain of IC development with a focus on the automated generation of digital RTL designs. The section also shows how this thesis delivers an approach based on Model Driven Architecture to realize these generators.

Model Driven Architecture relies on models, their Metamodels, and Model-to-Model transformations. An important consequence of this is that all approaches inspired by Model Driven Architecture are generally independent of languages and do not need to define languages and their semantics to achieve their generation goal.

This language-independent and Metamodel-based approach to the automated generation of digital designs is however not the only path to developing more powerful code generators: This chapter presents work related to the primary goal of this thesis, i.e. the development of methods for the automated generation of digital designs on the abstraction level of RTL design. These approaches, while aiming for the same goal, share the common property that they rely on custom languages defined for the development of generators.

This chapter first introduces the generation and configurability features available in Hardware Description Languages (HDLs) that are currently used by the industry and the established standard for digital design. It then introduces the concept of Embedded Domain-Specific Languages (EDSL), which is an approach to the development of custom languages that is heavily applied to develop custom, language-based code generation methods. Eventually, this chapter will present two important stages of evolution in the development of generator languages as EDSLs:

First, it shows how the EDSL concept has been applied to significantly extend the configurability of existing industry standard HDLs with their underlying simulation semantics.

Second, it introduces the family of so-called Hardware Construction Languages (HCL), also referred to as Hardware Generation Languages (HGL). Those are a group of Embedded DSLs that are developed from the ground up with a focus on hardware generation and the need for productivity during generator development.

3.1. VHDL, Verilog, and SystemVerilog for Configurable Designs

This section summarizes the capabilities of the standard HDLs VHDL, Verilog and SystemVerilog to provide generic, configurable designs. It introduces the methods available with examples and describes what is currently utilized in the industry. Based on this description, it provides a simple example highlighting the limitations of the features available to today's HDLs.

3.1.1. Preprocessors

Verilog and SystemVerilog support preprocessing. It is standardized using the terminology of *compiler directive* as part of the IEEE 1364-1995 Verilog standard and then further extended with the subsequent Verilog standards and the combined IEEE 1800 for Verilog and SystemVerilog [10, 74]. In terms of feature set, capabilities supported by recent versions of Verilog and SystemVerilog are a more limited version of the preprocessors implemented by compilers for C and C++ and specified by their language standards [57].

Directives for simple preprocessing needs are commonly used in the industry. Typically, they are utilized to enable different representations for the design and synthesis of Verilog or SystemVerilog models. A common industry example of this is the handling of analog types, which are often mapped to the language's logic type for synthesis while being represented as a real-valued type for simulation. Typically, this is achieved with simple `` `ifdef `` or `` `ifndef `` blocks and additional defines which are introduced into the design via the tool or compiler interface (e.g. through the TCL scripting interface provided by the tools). Listing 3.1 contains a Verilog example of such a pragma, introduced to resolve delta-cycle issues in mixed-language simulations.

```
1 assign
2 `ifdef SIMULATION
3   #1ps
4 `endif
5  b = a;
```

Listing 3.1: Example of an `` `ifdef `` pragma to alter simulation behavior

Another common application is the use of *include guards* as shown in Listing 3.2. Here, an entire file's content is wrapped by an `` `ifndef `` statement that can only be true once for every compilation. This pattern, common also in C and C++ development, ensures the content wrapped in the `` `ifndef `` will be included at most once, regardless of how often the `` `include `` preprocessor directive is encountered.

The more advanced macro shown in 3.3 is, for example, used to provide a shorter syntax for consistent logging of messages with the filename and line number of where they originate [57].

These common use cases are nowhere near the idea of hardware generation. Creative combina-

```
1 `ifndef _UNIQUE_FILENAME_V
2  `define _UNIQUE_FILENAME_V 1
3  ...
4 `endif
```

Listing 3.2: Example of an `ifndef pragma combined with a `define as include guard

```
1 `ifndef MSG
2  `define MSG(msg) \
3     $write("[%0t] %s:%d: ", $time, `__FILE__, `__LINE__); \
4     $display(msg);
5 `endif
```

Listing 3.3: Consistent logging of error messages with date and source location [57].

tions of `define, `undef, `include and others can be utilized to achieve preprocessor support for constructs such as repetition, lookup tables, or multiple definitions of similar modules with a different set of ports. Building generators like this is however very limited, hard to read and debug, and will inherently come with incompatibilities in many EDA tools. The fact that there is not even an `if in the SystemVerilog specified directives also clearly illustrates that the *compiler directives* defined in the SystemVerilog standard are not intended for use in hardware generation [57].

The VHDL language is no different here: while the preprocessor specified in the latest VHDL standard IEEE 1076-2019 supports so-called conditional compilation directives, there is to date virtually no support from EDA vendor side for this extension. While VHDL's conditional compilation directives are more powerful than those defined for SystemVerilog, the same limitations apply: today's HDLs are not a tool that was designed for building highly flexible, re-usable hardware generators [76].

3.1.2. Parametrization and Language Generation Constructs

Both VHDL and SystemVerilog support parametrization. In the SystemVerilog standard and language, parametrization can be achieved by means of *Parameters*. In VHDL, the equivalent language construct is called *Generic*. The following assessment refers to both concepts as parameters.

It is common to use parameters for configurable properties of design modules. Common examples of parameters are:

- Boolean flags that enable or disable some features of a module in some of the instantiations.

- String parameters that contain a module identifier used for example for logging purposes during simulation.
- Integer parameters that define for example the width of a datapath, the size of internal memories or register banks, or the number of instantiated submodules.
- Address parameters that define the absolute offset of the memory addresses served by a module's bus interface.

The nature of these common parameters already shows module parametrization is a feature that is better suited for the generation of generic hardware. The parameters can be used in various places of the design, for example, to replace constant literals in logical operations or as part of type declarations to modify the width of a wire or module port. Moreover, they can be passed to submodules as their parameters.

In combination with parameters, the `generate` construct supported by both SystemVerilog and VHDL can be used to make the internals of designs configurable. Generate blocks support loops where the number of iterations is configured by a parameter, they provide support for `case` statements with which alternative hardware options can be generated depending on the value of a certain parameter and for simple `if elseif else` blocks. These constructs can in turn be nested inside a generate block.

Figure 3.4 shows a SystemVerilog example of a module that uses a `generate` block to support a configurable amount of channels. Depending on the number of supported channels, the number of `single_channel` submodules that will be instantiated differs. Individual channels can be mapped to individual bits or subarrays of the inputs using the loop variable `i`.

3.1.3. Limitations

Generate and its feature set is however still very much limited in both VHDL and SystemVerilog. One of the fundamental limitations is that it is not possible to introspect and analyze part of the design that the `generate` statement is embedded in. This means that in the context of the `generate` statement, the only information that can be utilized are the parameters themselves and the relationships between them.

Moreover, the `generate` blocks have to be part of the module body (architecture body in VHDL). It is therefore not possible to conditionally add or remove ports or generate a flexible number of module ports.

The most noteworthy limitation in this context is the inability to take more complex input parameters. For both Verilog and VHDL, the parameters supported are restricted to simple types by design tools. Complex models, with well-defined schemata or Metamodels are not supported to parameterize designs at all.

```
module multichannel_ip
 #(parameter CHANNELS=8;
   parameter DATA_WIDTH=32; )
 (clk, data, irq);

 input  clk;
 output [CHANNELS-1:0][DATA_WIDTH-1:0] data;
 output irq;

 ...

 // vector dependent on the number of channels
 wire [CHANNELS-1:0] irq_sources;
 assign irq = |irq_sources;

 // Genvar to use in the for loop
 genvar i;
 generate
  for (i=0; i<CHANNELS; i=i+1) begin
    single_channel channel_i (
      .clk      (clk),
      .irq      (irq_sources[i]),
      .data     (data[i]),
      ...
    );
  end
 endgenerate

endmodule
```

Listing 3.4: Example of a generate statement in SystemVerilog

3.1.4. Conclusion

We have seen that language features of VHDL and SystemVerilog make the construction of configurable HDL modules possible to some extent. The preprocessors included in these languages are very limited, and while theoretically possible, the use of the preprocessor for building generators will lead to extremely clumsy syntax and difficult readability. Moreover, there is very limited tool support for more complex constructs and all configuration needs to be passed in via individual, primitive parameters that need to be combined into files that contain a set of `define` statements or passed in individually through the interfaces of the EDA tools interpreting the code. These approaches are fundamentally less capable than even the simple script-based generation approaches described in Section 2.3.1.

The parameterization of modules in combination with generation constructs built into the HDLs is a more structured and well-supported approach to building generic and configurable HDL designs. They come with some limitations and are fundamentally unable to support any kind of introspection into the design in which they are utilized. The clear strength of a parametrization-based approach is its integration into the HDL language standards and tool support throughout the EDA ecosystem.

Even with parametrization and built-in generation constructs, there is a clear and well-defined limit to the capabilities of the generators - many things can simply not be achieved. What becomes clear is that both VHDL and SystemVerilog as languages were designed to describe one instance of a design and provide some configurability as a second thought. The degree of flexibility and configurability that will be detailed in the rest of this chapter and that is eventually achieved by the Model Driven Architecture approach pursued in this thesis is simply not possible with traditional HDLs.

3.2. Enhanced Preprocessing and Script-based Generation

To address the limited capabilities of the HDLs commonly used in industry, many methods that are based on preprocessing or describing the HDL code in different languages have been developed. The most primitive way of achieving this is described already in Section 2.3.1. Here, scripting languages such as Perl and Python are utilized to do the preprocessing [16, 63]. This brings a significant advantage over the preprocessors defined for the HDLs themselves: the capabilities and readability of the scripting languages are well-supported and understood by a wide range of developers.

More advanced methods to do this introduce template engines that allow developers to primarily develop in the HDL of their choice and to only interleave their code with *preprocessor directives* where needed – a significant readability improvement over scripts with many print statements producing static elements of the target view. A good example of this is Genesis2 [93]: the code generation framework utilizes Perl and a Perl template engine to interleave Perl preprocessing logic with Verilog files.

Special `*.vp` files can contain Verilog code that is interleaved with Perl code. Listing 3.5 contains an example for such a file provided in the original work creating Genesis2. The Verilog module is interleaved with Perl commands and control statements. The framework then transforms and executes the Perl code and generates `*.v` Verilog files from it. For example, Lines 15-18 span a Perl `for` loop that will print the content of Lines 16 and 17 multiple times, depending on the value stored in `$N`. For each of the printed lines, Perl expressions enclosed in `'` will be evaluated and replaced by their evaluation result [93].

The template engine described here is very similar to what is part of the Metamodeling-based generation flow that Section 2.3.3 describes. There is however one significant difference: the generation flow sketched here does not utilize the sophisticated Metamodels and the Metamodel-based generation common for modern Metamodeling-based generation flows.

```
1 //; use POSIX ();
2 module 'mname()'
3  (input logic ['$N-1':0] pp['$N-1':0],
4   output logic ['2*$N-1':0] sum,
5   output logic ['2*$N-1':0] carry
6   );
7   //; my $height = $N;
8   //; my $width = 2*$N;
9   //; my $step = 0;
10  // Shift weights and make pps rectangular (insert 0s!)
11  logic ['2*$N-1':0] pp0_step'$step';
12  assign pp0_step'$step' = {{('$N'){1'b0}}, pp[0]};
13
14  //; for (my $i=1; $i<$N; $i++) {
15  logic ['2*$N-1':0] pp'$i'_step'$step';
16  assign pp'$i'_step'$step' = {{('$N-$i'){1'b0}}, pp['$i'], {'$i'{1'b0}}};
17  //; }
18  ...
19  assign sum = pp0_step'$step'['2*$N-1':0];
20  assign carry = pp1_step'$step'['2*$N-1':0];
21 endmodule : 'mname()'
```

Listing 3.5: Example of Script-based Preprocessor from the Original Research Developing Genesis2 [93]

One limitation this method shares with the Metamodeling-based generation flow and with the development of configurable designs only relying on the features of VHDL and SystemVerilog is the inability to introspect into the generated design as part of the generation flow. It is common for both approaches that the design is generated in the target HDL and not used during the generation flow but only afterward. Moreover, these approaches suffer from exactly the issues that were described as Generator Gap and they even show up in a more emphasized manner due to the lack of the structure that Metamodeling adds to the generation flow.

3.3. Embedded Domain-Specific Languages for Hardware Design and Generation

A Domain-Specific Language (DSL) in general is a programming language that is tailored to one specific domain or task. Popular examples of DSLs are Structured Query Language (SQL) (used to efficiently describe operations on relational databases), Regular Expressions (used to develop powerful string matching operations), the YAML markup language (used as a language that was specifically designed for entering and reading hierarchical, structured data), or JSON (an object notation format used for serialization and deserialization of object trees that is based

on a subset of the JavaScript language) [35].

YAML (a full-custom DSL) and JSON (an Embedded DSL) are good examples to show the differences between a full-custom DSL and an Embedded DSL:

- A full-custom DSL on is a language that was specifically designed for a certain task and comes with its own syntax and semantics, parser, libraries, and potentially runtime environment. YAML has a syntax that is perfectly tailored for readability, simplicity, and ease of use.

- An Embedded DSL is embedded on top of an existing language and utilizes the syntax, semantics, parser, and runtime environment of this language – it only needs to extend the semantics and provide a library. The example of JSON has a syntax that is fully compatible with the syntax supported by the JavaScript language. Any JSON code can be easily parsed and evaluated with a library running as part of the JavaScript interpreter without a need to develop custom parsers and runtime environments.

There are several advantages of Embedded DSLs in comparison to full-custom DSLs:

- Lower Implementation Complexity: The development of languages, parsers, and runtimes is one of the most challenging disciplines in software engineering. It consumes a lot of development resources and high levels of skill are needed to develop proper, maintainable languages that do not suffer from faulty language design. When developing Embedded DSLs, the resources can be focused on the extensions the DSL provides and on library features.

- Flat learning curve: Learning Embedded DSLs is easier, especially if the host language is well-known and has a large user base to support. Existing resources can be helpful and allow the user to focus on the domain covered instead of on the language.

- More powerful ecosystem: the strength of a language significantly depends on the ecosystem surrounding it. For Embedded DSLs, users can make use of the development environment and tooling (debuggers, profilers, IDEs, ...) available for the host language.

- Availability of all Host Language Features: Embedded DSLs just add something on top of the host language. In addition to using these features, semantics, and libraries, it is possible to access the full feature set of the host language. This is not limited to the language itself with the supported language patterns such as functional, object-oriented, or procedural programming. It also extends to existing libraries already available for the host language ecosystem.

These advantages are very significant. In the context of hardware generation, the availability of all host language features is arguably the biggest advantage as it provides virtually unlimited features including the application of software design patterns and libraries, the use of scientific and mathematical libraries, or the extraction of design properties for analysis. The relevance of

Embedded DSLs for hardware generation also shows in the related work identified as relevant to this thesis. All relevant research in the field, ranging from the late 1990s to today, relies on Embedded DSLs.

3.4. Traditional HDLs as Embedded DSLs

As a natural successor to the script-based approaches presented previously, the first approaches that ported traditional HDLs onto other host languages appeared at the end of the 1990s. The most notable candidates in this field are JHDL (1998) [14], Verischemelog (1999) [16], and MHML (2000) [17]. A key characteristic of these languages is that they all constitute a *translation* or *porting* of existing HDLs onto a host language while keeping the event-driven simulation semantics of the languages that inspire them.

In the context of this chapter, the Verischemelog approach is described in more detail. It is porting the Verilog language as an Embedded DSL onto the Scheme language (a LISP dialect). While LISP-based languages nowadays are rarely used and therefore do not provide many of the above-mentioned benefits of Embedded DSLs, one thing makes the example particularly well-suited: almost all language constructs in Verischemelog are identical to Verilog. Verischemelog thus provides a language that is structurally and semantically almost identical. At the same time, it turns it into an Embedded DSL and can be used to show the capabilities of Embedded DSLs and to highlight the advantages over preprocessing and scripting-based approaches.

```
1 // Half adder with prop. delay 10
2 module half_adder(b1, b2, s, c);
3   input b1;
4   input b2;
5   output s;
6   output c;
7   and #10 anonl(c, b1, b2);
8   xor #10 anon2(s, b1, b2);
9 endmodule
```

```
1 ;; Half adder with prop. delay 10
2 (defmodule half_adder
3   (interface (input b1 b2)
4              (output s c))
5
6
7   (and (10) (c b1 b2))
8   (xor (10) (s b1 b2))
9 )
```

Listing 3.6: Verilog (left) and Verischemelog (right) Example of a Half-Adder [16]

Listing 3.6 shows two models of a half adder provided in both Verilog and Verischemelog. This example is representative of the very similar structure between the two languages that also expands to other constructs such as processes, sensitivity lists, logging, wire definitions, and other language constructs. On the surface, this may just seem like a re-implementation of Verilog that was more simple to implement as it relies on an existing language parser. It however comes with a key difference: on the right-hand side, the model lives inside the programming language environment of Scheme and can therefore use all language features.

Figure 3.7 revisits the example from Listing 3.4 and implements it in Verischemelog. For this purpose, no kind of `generate` statements or Verilog generics are used. Instead, the functional

patterns of Scheme and general-purpose programming are utilized.

```
1  (define generic_multichannel_ip
2   (lambda (CHANNELS DATA_WIDTH)
3        (defmodule multichannel_ip
4         (interface (input clk)
5                    (input (($ DATA_WIDTH)) data)
6                    (output irq))
7
8         ... ;; further toplevel logic
9
10        ;; vector dependent on the number of channels
11        (wire (($ CHANNELS)) irq_sources)
12        (ror irq irq_sources)
13
14        ;; the $$ sign puts the list of elements returned
15        ;; by iterate into the parent module definition
16        ($$
17         (iterate CHANNELS
18           ;; Genvar to use in the for loop
19           (lambda (i)
20             (single_channel
21               'clk
22               (: 'irq_sources i)
23               'data
24               ... ;; further ports of single_channel
25             )
26           ) ; end lambda
27         ) ; end iterate
28        ) ; end $$
29       ) ; end defmodule
30  )
31 )
```

Listing 3.7: Implementation of the Code from Listing 3.4 in Verischemelog

The main aspect here is that this listing does not consist only of one `defmodule` call. Instead, it defines a lambda function on toplevel that will, when called with the generic parameters of the module, return the `defmodule`. This means it is possible to get a certain instance of the module `multichannel_ip` by calling the generator `generic_multichannel_ip` with the arguments `CHANNELS` and `DATA_WIDTH`. This call is what replaces the call arguments in the code with their value. Lines 16-28 are the equivalent of the SystemVerilog `generate` in Listing 3.4. Here, the functional equivalent of a `for` loop, an `iterate` with a lambda representing the "generate loop body" is providing the number of modules defined by the generic.

This section has demonstrated that it is possible to implement Hardware Description Languages as Embedded DSLs. These Embedded DSLs can be used to describe generators for configurable hardware, not just to describe a single, somewhat generic, and configurable instance of a circuit. With this approach, the development lo longer happens in an environment with two different languages. Instead, there is one language and one language syntax only and the benefits of Embedded DSLs fully apply.

An additional important benefit is that all hardware described as part of the Embedded DSLs lives as an object inside the language of the generator environment. With template engine-based code generation, it is just text of a different programming language. Embedded DSLs remove the separation between generation and analysis of the generated hardware. It is now possible to perform introspection to analyze and alter elements of the hardware as it is generated.

3.5. Synthesis-centric Languages as Embedded DSLs

This section discusses the field of Hardware Generation Languages (HGLs). In related work, this family of languages is often also referred to as Hardware Construction Languages (HCLs). Moreover, some publications and languages refer to these approaches as Hardware Description Languages (HDLs). The use of the term HDL emphasizes that this family of languages can be used as a replacement for the common HDLs VHDL and SystemVerilog for the task of developing synthesizable hardware descriptions – it does however not correctly capture the key aspect of these languages: their focus on describing generators for different configurations of hardware instead of describing single instances. To emphasize the focus these languages put on providing highly flexible generators for synthesizable designs, this thesis consistently uses the terminology Hardware Generation Language (HGL) to refer to these languages.

The research done in the field of Hardware Generation Languages is strongly based on the idea of developing generic design generators instead of providing single instances of a design. The languages are the natural successors to the languages described in the previous section – the differentiation between Hardware Description Languages implemented as Embedded DSLs and Hardware Generation Languages implemented as Embedded DSLs is however not clear.

In this research, the following as key characteristics for EDSL-based HDLs are assumed: An HDL based on EDSLs is a language that was developed with the main objective to provide or extend the capabilities for configurability, parameterization, and generation of an existing Hardware Description Language. The Hardware Description Languages VHDL, Verilog, and SystemVerilog which are commonly supported by the EDA industry were initially developed and used exclusively for simulation purposes. The semantics they use to describe hardware is the semantics of event-driven simulation. These languages have later been adapted for synthesis purposes. While design engineers think in the semantics of hardware, often referred to as *implementation thinking*, they need to map this mental model onto the event-driven simulation semantics of HDLs. Synthesis tools then have to infer the design intent from these descriptions. EDSL-based HDLs use the same underlying simulation semantics and event-driven paradigm of their parent HDLs and suffer from the same problem.

For Embedded DSL-based HGLs, this key characteristic does not hold true. In this research, HGLs are defined as languages that are not inspired by the simulation semantics of VHDL and Verilog. The following two characteristics are therefore at the center of those languages:

1. They use the semantics of hardware and the thinking in terms of hardware design as the basis and semantic model.

2. They are built for generation and meta-hardware descriptions. HGL code is not a model of the hardware but an executable program that generates a description of the hardware when executed.

3.5.1. Chisel and SpinalHDL

The Hardware Generation Languages Chisel and SpinalHDL are today the most important representatives in the field of Hardware Generation Languages. Chisel was originally developed as part of a research project at UC Berkeley. It was built to allow hardware designers to write high-level, synthesizable hardware descriptions [58].

Chisel is an Embedded DSL based on the Scala programming language, a multi-paradigm language with strong support for functional and object-oriented programming that is compiled to run on the Java Virtual Machine (JVM). The use of Scala as the host language allows Chisel to profit from the strong interoperability with libraries and compatibility with tools in the Java ecosystem. A second benefit of Scala is its good support for functional and object-oriented programming, making it an ideal basis to build an Embedded DSL. In fact, the Scala language has even been designed with the Embedded DSL use-case in mind [58].

The initial version of the Chisel language was released in 2012. The accompanying research demonstrated that Chisel is suited to build complex and flexible hardware generators, providing a high degree of efficiency and a level of abstraction superior to SystemVerilog. It also demonstrated that Chisel was able to emit RTL designs that were competitive in terms of their area and power usage [58].

The potential productivity benefits of the Chisel application also made the language interesting for industrial applications. The initial version of Chisel was however lacking the necessary features for state-of-the-art industry designs.

Essential shortcomings of the initial version of Chisel were the lack of support for multiple clock domains within one module, a lack of support for falling edge clocks and active low resets, a lack of clock enable support, and missing support for blackbox modules to support the embedding of legacy circuits described in standard HDLs. Moreover, several shortcomings in convenient reset handling and syntactical inconveniences limited the benefits of the language. Another important deficit of the initial version of Chisel was its failure to strictly enforce hardware semantics: `when` blocks Chisel uses to describe multiplexer structures did initially not ensure that all values were always described. It was therefore possible to generate RTL that contained asynchronous signals

which were not assigned in every way. This behavior was basically introducing event-driven simulation semantics into the language. Eventually, the failure of the Chisel project to address these issues in a timely manner resulted in the creation of SpinalHDL [71].

After the fork of SpinalHDL from Chisel, both languages have been developed in parallel and are used in academia and to some extent in industry. It is important to note that all of the shortcomings identified by the SpinalHDL project in 2016 are resolved and no longer apply to the latest version of Chisel3. The differences between the syntax, feature set, and library of SpinalHDL and Chisel have been growing over time. In the scope of this thesis, only Chisel will be introduced as it is the far more popular language, equally practical for use in both academic and industrial contexts and the subject of more ongoing research and development. The following describes the internal semantics of Chisel and the backend infrastructure and architecture.

3.5.1.1. The Semantics of Chisel

Semantically, Chisel is a design-centric Hardware Generation Language. It is by construction free from any event-driven simulation semantics. This means that every line of code written in Chisel either describes the design or contains generator logic responsible for configuring and customizing the generated design.

When Chisel code is executed, a hardware design description is built. This construction of the design has several important characteristics:

- Due to the absence of any simulation semantics, all stateful elements such as Registers and Memories always need to be explicitly instantiated and connected and are never inferred. Generator Code that would lead to the incomplete definition of multiplexers due to missing assignments of combinatorial logic values will not run in Chisel, i.e. no hardware description for simulation or synthesis is generated.

- Connectivity is described using wires and connections. There are no signals or signal semantics, just wires in the constructed hardware model. Consequently, there cannot be any *assignment* of values. The `:=` operator, which is sometimes referred to as the assign operator in the Chisel documentation and publications is actually a pure connection and connection modification operator.

- Combinational logic is defined using wires and expressions. In this context, every wire and port (their generalization in the Chisel core is referred to as a node) must have a driver connected to it. Undefined states thus do not exist.

The description of combinational logic in Chisel is quite intuitive and will not be further detailed here. An essential aspect of it, the creation of multiplexers is however intertwined with the connectivity creation of the Chisel language. Both aspects will be described in the following.

As mentioned before, Chisel supports ports, literals, and wires and generalizes these elements as *nodes*. Connections between these nodes are created using the `:=` connection operator. The example in Listing 3.8 contains three nodes: `io.input`, `io.output` and `myNode`. Two out of those nodes need to have a driver inside the module (`io.output` and `myNode`). The example contains two noteworthy aspects:

```
1  class SampleModule extends Module {
2    val io = IO(new Bundle {
3      val input = Input(UInt(8.W))
4      val output = Output(UInt(8.W))
5    })
6
7    val myNode = Wire(UInt(8.W))
8    io.output := myNode
9
10   myNode := 255.U // connect a constant literal to the node myNode
11   myNode := io.input // disconnect driver 255.U connected to the node myNode
12                      // and connect io.input to the node MyNode
13 }
```

Listing 3.8: Sample of a Chisel Module with conflicting connection statements

First, the two connections created in Lines 10 and 11 seem contradictory as they would connect two drivers to the same node. This is resolved by the definition of the `:=` operator in Chisel. This operator is defined to do two things: First, it will also disconnect any other driver connected to the left-hand side node if there are any connections already. Second, it will connect its right-hand side driver to its left-hand side node instead. In the given hardware model, `io.output` will therefore be connected to `myNode` which in turn will be only connected to `io.input`, not to the literal.

(a) model after execution of Line 10 (b) model after execution of Line 11

Figure 3.1.: Hardware Models Generated by Chisel Listing 3.8

This behavior is illustrated by Figure 3.1, which shows the internal connectivity model the Chisel library builds from the executed code. Sub-Figure (a) shows the status of the internal model after the first connection in Line 10 has been executed. Sub-Figure (b) then shows how the connectivity model changes when Line 11. What happens is that the connection from Line 10 is removed again, resulting in a model that is functionally equivalent to a model where the

line of code is commented out.

To fully understand the semantics of Chisel, it is also important to understand the impact of the conditionals `when`, `elsewhen`, and `otherwise`. The Chisel `if` constructs behave similarly to the `if` constructs in SystemVerilogs `generate` blocks: they are evaluated at runtime of the generator and will statically decide whether a certain part of the generator will be run or not, i.e. whether a certain piece of hardware will be generated or not. The `when` constructs on the other hand are conditionals that will be included in the design. In other words, these constructs are responsible for generating multiplexers with conditions as their select signals. On the Chisel language level, the `when`, `elsewhen`, and `otherwise` blocks impact the design by modifying the behavior of all the connectivity operators `:=` used inside these blocks. If the connectivity operator is embedded in one or multiple of these conditional statements, it will no longer simply disconnect any driver from the left-hand side element and connect the right-hand side element as a replacement. Instead, a connectivity operator embedded inside one or more `when` statements instantiates a multiplexer. In other words, the `when`, `elsewhen`, and `otherwise` blocks are a convenient and flexible way to introduce multiplexers into the connection path.

```
1  class SampleModule extends Module {
2    val io = IO(new Bundle {
3      val input = Input(UInt(8.W))
4      val output = Output(UInt(8.W))
5    })
6
7    val myNode = Wire(UInt(8.W))
8    io.output := myNode
9
10   myNode := 255.U
11
12   when(io.input < 128.U) {
13     myNode := io.input
14   }
15 }
```

Listing 3.9: Sample of a Chisel Module with Multiplexer Connection

Listing 3.9 is a modified version of Listing 3.8 where the connectivity operator for the connection from `io.input` to `myNode` is embedded in a conditional when block. Inside this when block, the connectivity operator will exhibit the following behavior: It will disconnect the current driver (`255.U` in the example) from `myNode`. It will then create a new multiplexer in the design and connect the output of this multiplexer to `myNode`. It will connect the right-hand side of the connection operator to the true path of the multiplexer and the old driver it had disconnected to the false path of the multiplexer. Eventually, it will generate hardware to evaluate the boolean truth value for the and-combined condition of all when clauses the connection operator is embedded into (in this example just one clause and one condition). This condition is then connected to the select input of the multiplexer. Figure 3.2 shows the hardware generated

by this example: both `io.input` and the literal `255.U` are connected to a multiplexer. The select of the multiplexer is connected to a less-than comparator (`lt`) which compares the input `io.input` to the constant `128` provided in the Chisel code.

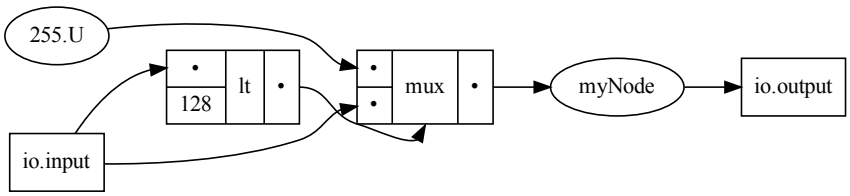

Figure 3.2.: Hardware Models Generated by Chisel Listing 3.9

If the code is modified from Listing 3.9 and the statement in Line 10 is commented out, this will remove the connection between the literal `255.U` and the input of the multiplexer created by the `when` block in combination with the assignment in Line 13. The input node of the multiplexer would therefore not be connected to any driver, violating the Chisel rule that every node needs to have a driver and the underlying hardware semantic. Running this Chisel generator will fail with an exception highlighting the issue and not generating HDL output.

The last important point regarding the semantics of Chisel is its stateful elements. For every register that is created in Chisel, the data output node and the data input node are connected. Following the semantics of the connectivity operator `:=`, an assignment to the register that is not wrapped inside a `when` statement will disconnect the data output from the register and attach the right-hand side of the assignment instead. The implications of this definition and the Chisel semantics described above are illustrated in Listing 3.10, which is a modified version of the last example in Listing 3.9.

For this example, the wire in Line 7 has been replaced and the connection created in Line 10 has been commented out. Because of the replacement of the `Wire` with a `Reg` and because the implicit connection between register output and register input is automatically created for every register, this example generates HDL despite the missing default assignment. The assignment in Line 13 removes the data output of the register from its data input and attach it to the newly created multiplexer instead.

3.5.1.2. Advantages and Limitations

One of the key advantages of Chisel lies in its relative simplicity compared to the traditional, industry-standard HDLs. A big part of this comes from Chisel's semantic clarity: It is fundamentally a design language based on the underlying hardware semantics and free from event-driven or simulation-based concepts. It does not infer stateful elements and every Chisel statement ultimately creates hardware and connections between this hardware. Moreover, it has a

```
class SampleModule extends Module {
  val io = IO(new Bundle {
    val input = Input(UInt(8.W))
    val output = Output(UInt(8.W))
  })

  val myReg = Reg(UInt(8.W), init: 0) // replace the wire with a register
  io.output := myReg

  // myReg := 255.U // comment out this line

  when(io.input < 128.U) {
    myReg := io.input
  }
}
```

Listing 3.10: Sample of a Chisel Register and Port (Nodes) without Driver

very concise syntax, making it easy to write and read generator code.

Chisel has gained high popularity over the last years, with plenty of research work, open source IP components, learning materials, and a vivid community developing based on it. Recurring conferences and some limited industry adoption also have helped the Chisel development.

We identify two main, high-level drawbacks of the Chisel approach:

Complex Host Language In terms of the initial learning curve, a significant drawback of the Chisel approach becomes visible. Chisel suffers from the relative obscurity of the underlying host language Scala: none of the common rankings published for programming languages places Scala in the top 20 languages [100, 102]. Scala is a complex multi-paradigm language that has a significantly higher entry barrier than other general-purpose languages. Scala is not a language that is already part of science and engineering university curricula or commonly used for scripting and general-purpose programming and automation. When it comes to available libraries, Scala code can easily access the Java ecosystem [31]. A set of libraries that is particularly useful for automation in the field of IC design are however libraries for numerics, data science, and artificial intelligence. These libraries are easy to access from native compiled languages and Python, which typically acts as the front-end language for the AI and ML ecosystem. Bindings to the Java ecosystem are only available in a limited manner. In Section 3.3, we identified a flat learning curve and availability of libraries benefits of Embedded DSLs. Given the properties of the Scala language, this benefit does not fully apply to Chisel [58].

Lack of visible and clearly defined underlying Metamodel During the execution of Chisel code, the data structure is built from the Chisel language's instructions. This data structure is then processed and written out as Verilog for synthesis. The easiest path to understanding and conceptualizing Chisel is to understand the language definition. Following this learning path (getting the syntax and semantics of the language instead of learning the structure and semantics of the underlying model) makes the initial learning of Chisel easier, however, comes with a caveat: This learning path does not require an understanding of the model underlying the Chisel-generated data structures.

The *Complex Host Language* of Chisel and its properties, operators, and constructs – which may be unknown to the developer of Chisel code – are closely interwoven with the operators and constructs that the DSL puts on top of it, making it difficult for the user to understand what actually happens when the code is executed. The lack of a visible and clearly defined underlying Metamodel or more precisely the lack of awareness of this Metamodel makes it difficult to understand what the individual Chisel statements do to modify this Metamodel. Clear visibility on the existence of the Metamodel and the actions done by the executed generator code to this Metamodel helps the developer to understand that he is writing a generator instead of writing HDL code.

We consider two points as key features of EDSL-based Hardware Generation Languages:

1. The ability to perform introspection on the existing data structures describing the design *during* the execution of the generator.
2. The ability to apply transformations on the generated data structures.

These two features cannot easily be achieved due to the lack of a visible, clearly defined underlying Metamodel: Chisel was fundamentally designed as a language and not as a tool to define a model. The language specification hides the model underlying these data structures, making the development of introspection or transformation-based approaches more difficult. It is possible to perform introspection on generated structures during the execution of the generator and to apply any transformation to them. This however requires an understanding of the internals of the Chisel library and compiler. Such an understanding is not at the center of a language-based approach and is therefore difficult to obtain. Moreover, the data structures this will work on are however outside of the specification of Chisel and the developed code may fail in later versions of the language.

We also see the model and the understanding of the Metamodel as a central point to understanding the semantics of a language. The absence of this Metamodel in the Chisel learning path makes it difficult to separate the (very nice and clean) semantics of the Chisel language from the semantics of the target view (event-driven simulation language such as SystemVerilog) without a focus on the underlying generated model.

A further disadvantage of Chisel is the relatively low level of the underlying internal model that Chisel data is mapped onto. The section on the semantics of the language already described that Chisel code is basically broken down into a graph of nodes, with registers, combinational logic, and submodule instantiations as its key elements. Chisel notably does not contain common

higher-level constructs such as finite state machines that are part of the design thinking of hardware developers. This can be a significant limiting factor for transformations and analysis on the model built by Chisel code.

Assessment of Limitations by the Chisel Team Some of these drawbacks are also identified and acknowledged by the authors of Chisel [70]. They correctly note several limitations that all originate from a non-model-based, language-centric approach:

1. "Writing custom circuit transformers requires intimate knowledge about the internals of the Chisel compiler" [70]. We attribute this limitation mainly to the *Complex Host Language*.

2. "Chisel semantics are underspecified and thus impossible to target from other languages" [70]. We attribute this limitation to the *Lack of underlying Metamodel*.

3. "Error checking is unprincipled due to underspecified semantics resulting in incomprehensible error messages" [70]. We attribute this limitation to the *Lack of underlying Metamodel* with a formal, Metamodel-based description of legal model states.

4. "Learning a functional programming language (Scala) is difficult for RTL designers with limited programming language experience" [70]. From our point of view, this originates from the *Complex Host Language*.

5. "[C]onceptually separating the embedded Chisel HDL from the host language is difficult for new users" [70]. From our point of view, this originates from both the *Complex Host Language* and the *Lack of underlying Metamodel*.

3.5.1.3. FIRRTL

The developers of Chisel have addressed the shortcomings they identified with the introduction of FIRRTL (Flexible Intermediate Representation for RTL). FIRRTL is described by the authors as a Hardware Description Language that represents the elaborated Chisel circuit, having many syntactical and semantic similarities with the Chisel language, however not contain any of the metaprogramming capabilities provided by the Scala host language [70]. The FIRRTL code is what is generated from the Chisel internal data structures describing the hardware. From the point of view of this thesis, FIRRTL is an intermediate model with a well-defined Metamodel that describes an elaborated Chisel design. In other words, it is the hardware-centric model built by the execution of a Chisel generator.

FIRRTL does address the shortcomings originating from the *Lack of underlying Metamodel* we described in the previous section:

- It provides a clean platform for applying transformations.

- It helps developers to conceptually separate the Chisel language from the host language as they can see what the evaluated circuit looks like without the host language generator constructs, yet still in a syntax similar to that of Chisel.

- It can be directly generated without using the Chisel generator if a different path, absent of Chisel HGL has to be used to generate circuits.

While it can be clearly seen that FIRRTL is removing these shortcomings and making the Chisel flow more similar to the flow presented in this work, there are a few remaining limitations and problems with the approach:

- The FIRRTL model is not the internal model of the Chisel language. FIRRTL thus does not support the process of introspection on the Chisel model at runtime of the Chisel generator. The FIRRTL model is similar, yet not identical to the internal model of the Chisel language. It does therefore not significantly ease the process of introspection on the Chisel model right when the Chisel generator is executed.

- Development of generators in Chisel is still the development of generators in a language defined for Hardware Generation, it is not the development of a transformation or the explicit generation of a target data structure. This target data structure is then transformed into FIRRTL. The developer is however not necessarily aware of this first layer of data structure that is constructed when Chisel executes.

- Instead of relying on an Embedded DSL, FIRRTL again introduces a full-custom DSL for an intermediate artifact. FIRRTL therefore in turn negates some of the benefits of working with Embedded DSLs: it is once again required to develop parsers and code generators for a new language, resulting in an increase in complexity and maintenance overhead. This thesis demonstrates that the benefits that FIRRTL provides can be achieved with a model and a defined Metamodel in the environment of an existing general-purpose programming language. It is easy to conclude that the introduction of a Metamodel instead of the introduction of a custom language would have been beneficial.

The Metamodeling-based Model Driven Architecture approach presented in this thesis does not exhibit these limitations and has further benefits regarding its learning curve and the areas it can be applied in. In the remainder of this thesis, the approach will be presented.

4. Model Driven Architecture for Hardware Development

This chapter introduces the adaptation of Model Driven Architecture for Digital Hardware Generation provided by this thesis. In line with OMG's vision of Model Driven Architecture, the adaptation presented here defines multiple layers of models, ranging from a very high level of abstraction of the specification of a design down to a very low level of abstraction akin to an Abstract Syntax Tree (AST) of the generated target views.

This chapter introduces the layers of models used in the MDA approach for digital hardware design and describes their content and level of abstraction. It describes the Metamodels used for the modeling layers and puts the defined layers into context with the corresponding layers defined in the OMG vision for Model Driven Architecture. It also establishes the terminology used throughout this thesis and in further research based on this thesis.

It is important that this chapter provides descriptions of its Metamodels based on the M3 Meta-Metamodel of Infineon's proprietary in-house Metamodeling framework. None of the aspects described in this chapter depend on anything other than the M3 Meta-Metamodel. This marks an important difference to the approaches presented in Section 3: The definition of the Model Driven Architecture inspired approach is independent of a certain target language and the concrete implementation of a defined software library or framework. The notation used in this thesis to visualize the Metamodels is that of UML class diagrams. Models are also visualized using UML instance diagrams. Further, as the M3 notation relies primarily on the definition of meta-objects, their attributes, and relationships, the results provided here can be easily mapped onto other M3 formalisms such as the Ecore Meta-Metamodel at the heart of the Eclipse Modeling Framework [49].

Any details on how the models are read, transformed, and analyzed are deliberately omitted in this chapter as these are characteristics of the MDA flow that are specific to a certain implementation in a Metamodeling framework. Chapter 5 introduces this framework and describes these aspects.

4.1. Overview

Figure 4.1 shows the proposed adaptation of MDA for digital hardware generation. This approach conceptually follows the vision of Model Driven Architecture as introduced in Section 2.4.1. Similar to the three-layer approach of the initial OMG proposal for Model Driven Architecture, the concept introduces three layers of models, each with the Metamodels constraining them and defining their characteristics.

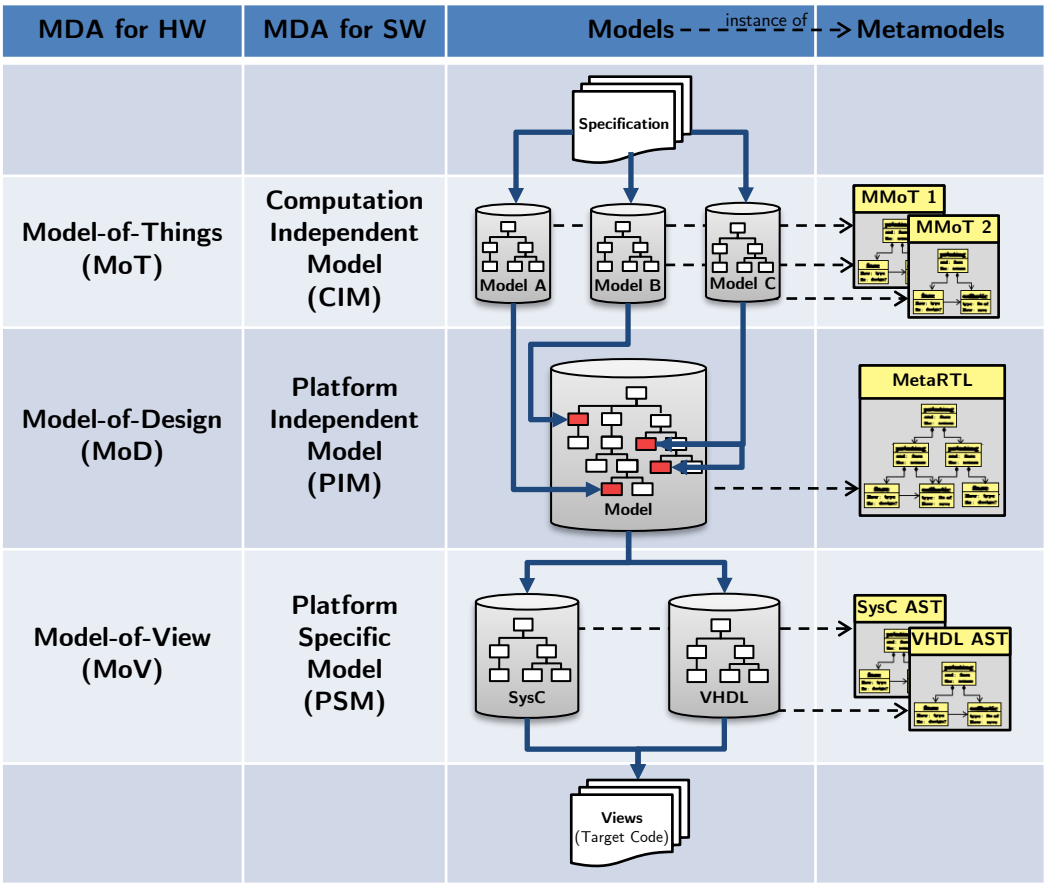

Figure 4.1.: Abstraction Layers in MDA for Hardware Generation – Models and Metamodels

For each of those three layers of abstraction, terms are introduced that are closely related to their purpose and level of abstraction of the corresponding model:

MoT The Model-of-Things corresponds to the Computation Independent Model (CIM). Its intention is to formally capture data from requirements and specifications. The MoT thus defines things, their attributes, and their relations to the intended functionality. In this context, functionality describes what the product has to provide, without including how the product is implemented, e.g. which algorithms and architecture are used. This makes the models *computation independent* in the terminology of OMG. In the context of Digital Hardware design, this can also accurately be referred to as *implementation independent*. Section 4.2 details the Model-of-Things and provides example Metamodels for this layer of abstraction.

MoD The Model-of-Design corresponds to the Platform Independent Model (PIM). Its goal is to define the architecture from the point of view of a hardware designer. The MoD and its Metamodel are the core components of the proposed methodology. Broadly speaking, a Model-of-Design describes the design on Register Transfer Level. It differs from modeling

languages based on event-driven semantics as it does not cover any simulation or synthesis artifacts of the views. It is no longer independent of the targeted implementation (i.e. microarchitecture) but independent of the language and platform targeted by the generation. The MoD and its Metamodel are detailed in Section 4.3.

MoV The Model-of-View corresponds to the Platform Specific Model (PSM). It is the least abstract model with a straightforward mapping to the target view. The level of abstraction of this layer can be compared to the abstraction of an Abstract Syntax Tree (AST) of the target view. The Model-of-View layer is detailed in Section 4.4.

In order to achieve the goal of this thesis (see Section 2.5), the framework needs to close the *Generator Gap* by making it easier to develop ever more powerful generators. A key element to this is the definition of suitable Metamodels for each of the respective modeling layers. There are several Metamodels on the MoT layer and MoV layer. For the MoD layer, this thesis introduces exactly one Metamodel.

MMoTs Every specification formalized in a Model-of-Things instance is transformed onto a set of components part of a Model-of-Design. On the Model-of-Things layer, several different Metamodels-of-Things are necessary depending on the design task. When generating several CPU cores with different RISC ISAs, the formalized description of these ISAs will use models of the same Metamodel. When a full CPU subsystem is generated, various peripherals such as timer, interrupt controller, or signal processing peripherals will require different Metamodels.

MMoD For the MoD layer, there is exactly one Metamodel that is related to the RTL and synchronous design abstraction. The name of this Metamodel-of-Design is MetaRTL. The MoD-layer and its Metamodel are the most important central cornerstone of the proposed adaptation of MDA for digital hardware design. For this reason, the MDA flow for digital hardware design is sometimes simply referred to as MetaRTL.

MMoVs The number of Metamodels on the MoV layer depends on the number of views targeted. As described above, the MoV is dependent on the language of the view that is generated. Therefore, the Metamodels-of-View is heavily influenced by the grammar of the view's language. In general, a separate Metamodel is used for every view. Section 5.4 of this thesis shows the View Language Description (VLD) approach and how even those MMoVs can be automatically generated.

The following sections will provide more detail on the individual layers of abstraction from top to bottom.

4.2. The Model-of-Things Layer

The Model-of-Things Layer formally captures data from requirements and specifications. The term Model-of-Things is used to clarify that the MoT defines things, their attributes, and their relations to the intended functionality. In this context, functionality describes what the product has to provide, without including how the product is implemented, e.g. which algorithms and architecture are used.

The equivalent layer in the terminology of OMG is the Computation Independent Model (CIM) which captures information such as business processes. In the context of Digital Hardware design, this can also accurately be referred to as *implementation independent*. It does not describe the inner workings of a potential implementation. Instead, it is a declarative approach to describing what the generated hardware shall implement.

Any available Model Driven Architecture flow will support a known set of Metamodels as its input on the MoT layer. When the MDA flow is applied to generate code, it will use one or multiple models of these Metamodels as input to the generation. These Metamodels therefore not only describe the MoT – they also describe and constrain the supported inputs for a given MDA flow.

The Metamodel used for the Model-of-Things layer is highly dependent on the type of hardware that shall be generated by the MDA flow. For example, there have to be different Metamodels-of-Things for different communication peripherals, bus fabrics, CPU cores, Accelerator IPs, and all other types of hardware. In the following, a very simple MMoT example is provided to illustrate the purpose and possible level of abstraction of the MoT layer.

4.2.1. Example MoT Layer for Digital Filters

This section provides an example of an MoT layer based on a hypothetical MDA-based generation flow for digital filters. This flow shall support the generation of hardware implementing digital filters which adhere to a recurrence relation specified in Equation 4.1. Input to the generation flow are the parameters b_i of this equation and the expected outcome is synthesizable RTL code implementing the filter.

$$y[n] = \sum_{i=0}^{N} b_i \cdot x[n-i] \tag{4.1}$$

Figure 4.2 shows the Metamodel-of-Things that formalizes the valid input formats for the MDA flow using Metamodeling methods. According to the pictured Metamodel, the FIR Model-of-Things describes a `FIRFilter` with a name `Name`. For every $b_i \neq 0$, an `Addend`

composition object is created, with an attribute `Instant` containing the index i. The attributes `ImpulseResponseReal` and `ImpulseResponseImag` contain the real and imaginary part of the complex number b_i. According to the Metamodel, every `Addend` has a mandatory integer `ImpulseResponseReal` and an optional `ImpulseResponseImag` (`ImpulseResponseImag` has multiplicity optional, i.e. `0..1`).

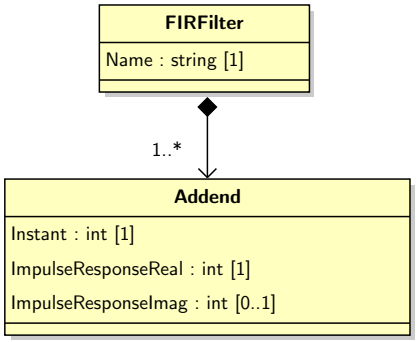

Figure 4.2.: Metamodel-of-Things for an MDA-based generator for FIR Filters

Any filter that adheres to the recurrence relation in Equation 4.1 can be expressed as a Model-of-Things instance of this Metamodel-of-Things. Equation 4.2 is an example of a 2nd order filter that adheres to the generic recurrence relation.

$$y[n] = 4 \cdot x[n] + 2 \cdot x[n-1] + 1 \cdot x[n-2] \tag{4.2}$$

This specification has to be provided as an MoT instance of the MMoT in Figure 4.2 so that it can be used for the MDA-based generation flow. Figure 4.3 shows the MoT of the filter from Equation 4.2. The attribute `ImpulseResponseImg` does not show up in the model since it is not needed for this instance of the filter (no imaginary component to the impulse responses). This complies with the Metamodel since `ImpulseResponseImg` has multiplicity optional, i.e. `0..1`.

Despite the simple structure of this example, a key characteristic of the proposed approach is visible here: The Model-of-Things contains only the specification. It is easy to imagine a set of microarchitectures to implement such a filter. For example, a simple one sample per cycle pipeline might be used (see Figure 4.10). In this case, the multipliers might be replaced by wires treating the coefficients as constants for the cost of reduced flexibility. Generally, the added chain can be replaced by an adder tree. Another microarchitecture would consist of an FSM and a multiply-accumulate unit. While these microarchitectures have different characteristics, they are both equally valid microarchitectures considering a specification that only describes parameters of the transfer function of the digital filter and does not contain any performance constraints such as throughput. The transformation from Model-of-Things to Model-of-Design can have many implementations resulting in significantly different architectures. The sample microarchitectures mentioned here also illustrate that there is not necessarily a one-on-one correspondence between elements in the Model-of-Things and elements in the Model-of-Design. In fact, the transformation from Model-of-Things to Model-of-Design bridges a large gap in the

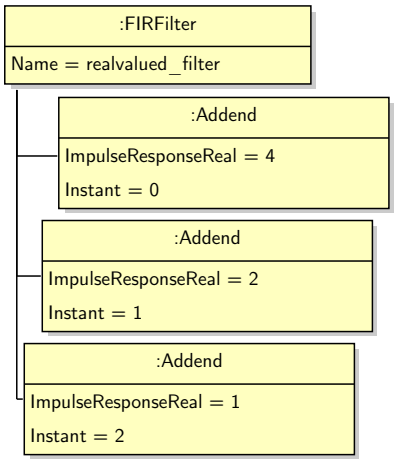

Figure 4.3.: Example Model-of-Things instance of FIR Filter MMoT

level of abstraction and the structure of the Model-of-Design does not typically resemble that of the Model-of-Things.

Moreover, the Metamodel-of-Things defined as input for the MDA-based filter generator is not the only feasible MMoT. The MMoT could for example also specify the expected intended frequency-domain characteristics and maximum deviation from these characteristics. In this case, the Model-to-Model transformation from Model-of-Things to Model-of-Design would first need to derive the coefficients b_i for a digital filter that meets the required characteristics. Chapter 5 will contain a detailed description of how an MoD realization of an MoT can be achieved as part of a Model-to-Model transformation.

4.3. The Model-of-Design Layer and the MetaRTL Metamodel

The Model-of-Design layer is the central element at the heart of the Model Driven Architecture for Digital Hardware Generation. For any execution of an MDA-based generator, there is exactly one instance of the MoD. This instance acts as a central representation of the design. All MoT inputs are processed and mapped onto the MoD using Model-to-Model transformations. After the transformations are completed, the MoD is the only source of information used to build one or more MoV instances used for the generation of the targeted HDL code.

4.3.1. Semantics of MetaRTL

The Metamodel of the Model-of-Design layer that was developed as part of this thesis is called MetaRTL. MetaRTL is a structural representation of the design on Register Transfer Level (RTL). It is important to note that MetaRTL semantically differs from the Hardware Description Languages (HDLs) SystemVerilog, Verilog, and VHDL which are commonly used for digital

hardware design and simulation on RTL.

The first key difference is that the model that is contained in an instance of the MetaRTL Metamodel describes one design in an elaborated manner. The elaboration is performed as part of the Model-to-Model transformation from MoT to MoD. Elaborated design means that re-usable modules are part of the Metamodel-of-Design, there is no concept of re-usable modules and instantiation in a MetaRTL model. Instead, the concepts of instantiation and re-usable modules are entirely located on the level of the MetaRTL Meta-model and the transformation from MoT to MoD.

The second key difference is the absence of simulation semantics in MetaRTL. Modern HDLs support the description of behavior, structure, and test and verification artifacts. It is possible to use these languages to describe a design on a structural level by instantiating and connecting existing building blocks. The implementation of these building blocks in HDLs is however always based on the languages' underlying event-driven simulation semantics. This is at odds with the mental model hardware designers have of the structures in their design: Registers, their clocking and reset behavior and the combinational logic between these registers have to be transferred to a description in event-driven semantics. The event-driven semantics modern HDLs come with a high degree of freedom to build event-driven models that have undesireable mismatches between their behavior in an event-driven simulation and the logical behavior of the circuits synthesized from the models, referred to as simulation/synthesis mismatch [15].

MetaRTL is a fully design-centric and structural model of hardware, based fully on the semantics of hardware and free from any simulation semantics. It contains structures describing the typical design patterns and elements part of hardware semantics such as registers with their clocking and reset properties, dataflow elements to describe the combinational logic as well as higher-level design patterns such as state machines, decoders, or lookup tables. Instead of relying on event-driven semantics, MetaRTL relies on simpler assumptions. First, MetaRTL is based on the timing assumption that all signals in a MetaRTL model stabilize before a relevant clock edge. Second, it assumes structural integrity of the design, i.e. the absence of combinational feedback loops.

The representation of MetaRTL models thus matches a hardware designer's thinking process and the mental model underlying digital design in the minds of digital design engineers. For engineers, the use of a model that matches their mental model and the underlying physical reality simplifies the design process: it frees designers from the mental load of having to map their design thinking onto the simulation semantics of HDLs – a mapping that also blurs the view onto the actual design patterns used for the creation of a digital design.

Moreover, it also eliminates the need to transfer back from simulation semantics to higher-level patterns in order to perform transformations and optimizations on a design. From a point of view of building generators and transformations, this is beneficial as it removes the need for difficult-to-implement heuristics that extract the higher-level design patterns. Chapter 5 will describe in detail how this simplifies the development of Model-to-Model transformations from MoT to MoD or between different versions of the MoD.

4.3.2. The MetaRTL Metamodel

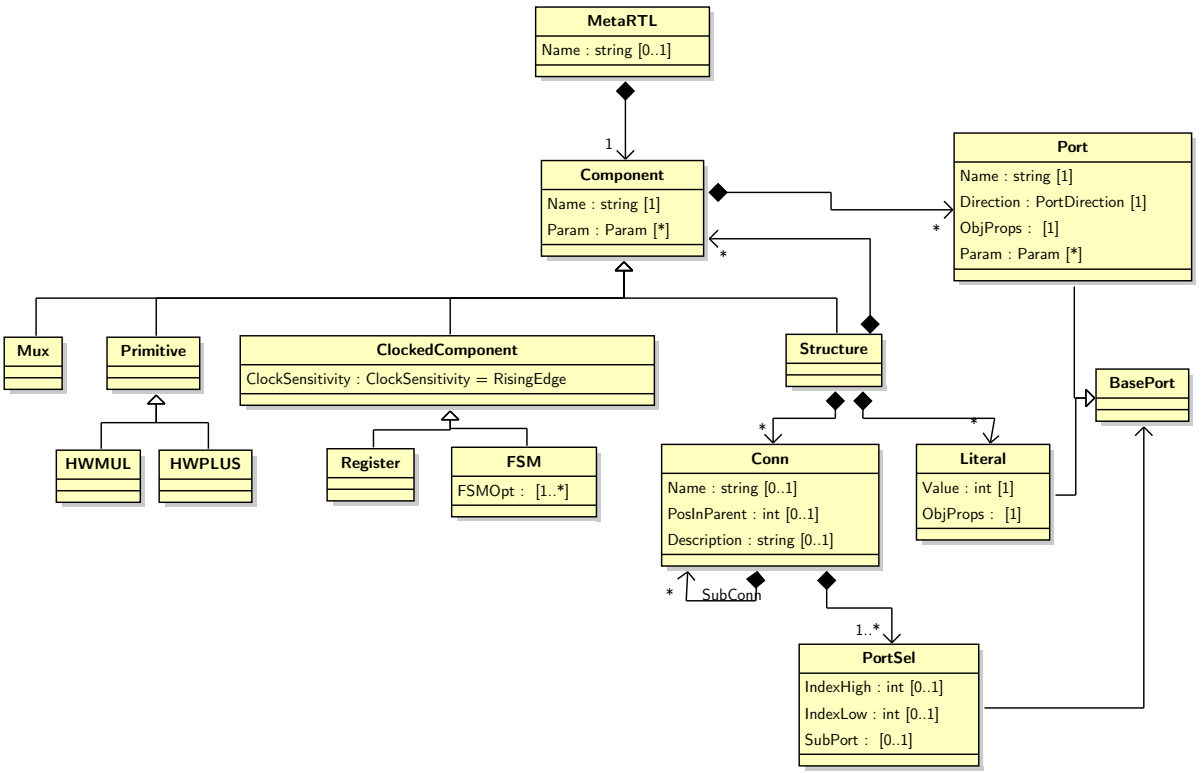

Figure 4.4.: MetaRTL: Metamodel-of-Design (simplified)

Figure 4.4 shows a simplified version of MetaRTL, the Metamodel of the Model-of-Design (MoD). According to the MetaRTL Metamodel, any design consists of a `MetaRTL` instance that contains exactly one instance of a `Component`: this instance is the root of the design generated by the MDA flow.

This component can be of different types as shown by the generalization relationship between `Component` and `Primitive`, `Mux`, the `Primitive` and its specializations `HWMUL` and `HWPLUS` as well as the `ClockedComponent` and its specializations `Register` and `FSM`. The simplified version shown in Figure 4.4 does not contain all component types supported by MetaRTL but just a selected set of a few components that can be used to construct an initial example. For now, it is important to note that all component types inherit directly or indirectly from the `Component` class in the Metamodel of MetaRTL and share its common properties and a set of boundary ports described by the composition of `Port` in `Component`.

The most important component of the MetaRTL is the `Structure`. The structure is the non-terminal MetaRTL element that allows for building hierarchical designs. The composition from `Structure` to the abstract base class `Component` in the Metamodel shows that the Structure in turn can instantiate several `Components`, i.e. further specializations of it such as further nested `Structure` instances, `Muxes` or further components. The `Structure` is what turns a MetaRTL design into a hierarchical design with connectivity. This connectivity is represented in the

Metamodel through the `Conn` class composed in the `Structure`. Connections themselves are complex objects that attach to either a `Literal` or a `Port` or a part of those. This attachment is modeled through the `PortSel` class in MetaRTL. These connection elements represent actual RTL connectivity between the `Structure`'s boundary ports and the boundary ports of the instances inside the Structure. In addition to describing instance-to-instance connectivity, any MetaRTL `Structure` can also contain `Literal` objects as containers for constant values that are used to drive the boundary ports of `Components` in a design.

This simple MetaRTL Metamodel is sufficient to describe the first designs on RTL level. As mentioned above, Figure 4.4 does not yet contain the full set of components required for digital design. The full set of components will be introduced and described later in this section. The diagram also does not yet contain a very important part of the MetaRTL Metamodel: The object properties used to describe type characteristics such as bit width and hierarchical structure and the interpretation of values on ports.

The following subsections use examples to explain several specializations of the MetaRTL `Component` to give a more detailed understanding of the Metamodel and its key characteristics and components.

4.3.2.1. Compatibility with all Naming Conventions

It is an important design goal of MetaRTL to be compatible with all kinds of different naming conventions. To guarantee compatibility with all naming conventions, the semantics of any model artifact must be independent of the Name attribute of the model artifact. This also allows users of MetaRTL to omit the names of Model-of-Design elements. These names will then only be introduced when a Model-of-View representation of the Model-of-Design requires it.

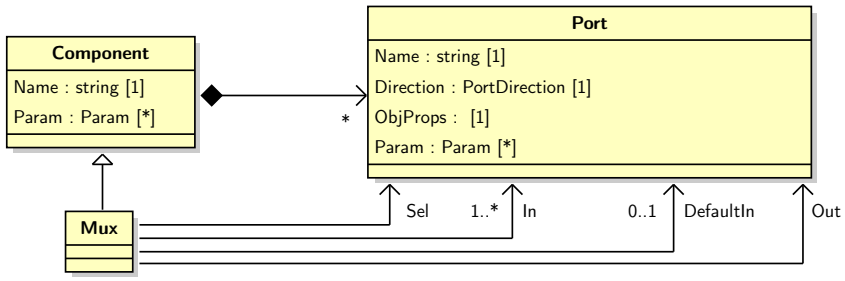

Figure 4.5.: Mux Component with associations to describe Port roles independent of their names

Figure 4.5 uses the `Mux` component which implements a hardware multiplexer as an example of the MetaRTL Metamodel implementation. Like all components, the Mux has a set of ports that describe its interface. For further Model-to-Model transformations, it is however important to know which roles these ports have. MetaRTL identifies 4 types of special ports for the Multiplexer:

- `Sel` for the select input of the multiplexer. This input determines which port is forwarded to the output of the multiplexer. In a correct model, the `Sel` association is required (Multiplicity of the Association is `1`).

- `In` for the inputs the `Sel` port selects from. This is one of the options that can be forwarded to the output. In a correct model, the multiplexer needs to have at least 1 `In` association, can however have arbitrarily many (Multiplicity of the Association is `1..*`).

- `DefaultIn` references the `In` port that is selected if the `Sel` value is not mapped to any input port. This is an optional attribute that may or may not exist in a real model. For some models (e.g. when the number of inputs is 2^n and the bit width of the `Sel` port equals n, the `DefaultIn` attribute does not affect the model.

- `Out` to define the data output of the multiplexer. This port will be driven with the input value that is selected by the `Sel` port. In a correct model, the `Sel` association is required (Multiplicity of the Association is `1`).

This approach allows to create the `Mux` component and any other primitive or complex components entirely without any names - the associations `Sel`, `In`, `DefaultIn` and `Out` defined above and the name of the associations are part of the Metamodel, they are not names in the model. They are what defines the roles of the ports and not their names. While this example is of course not meaningful, it would be possible to create a new Multiplexer and name the ports that are referenced by `In` as e.g. `MyOutputPort01`, `MyOutputPort02` and `MyOutputPort03` and the ports that are referenced by `Out` as `MyInputPort`. While this is of course not meaningful for understanding and debugging any design, it would create a perfectly fine circuit when used in the MDA flow. The more practical application of this feature is the omission of port names. In this case, the port names will be introduced by convention during transformations of the model.

By convention, the multiplexer's `In` ports will be mapped sequentially to the key space defined by the bits of the `Sel` port. The model can therefore be queried to identify which select value will choose which input port. It is furthermore possible to override this using the `Param` composition of the Port class. This composition allows to define one or more `Sel` values for which a given `In` port will be selected.

4.3.2.2. Registers

MetaRTL is semantically fully located on Register Transfer Level. As the model deliberately does not use any simulation semantics, the `Register` component of MetaRTL is the way to introduce registers into the design. Figure 4.6 shows the MetaRTL Register. Like all other `ClockedComponent`s, the `Register` required a `Clk` input port. In addition to that, an `En` port can be optionally provided to enable and disable the register. Resets of the MetaRTL `Register` component can be synchronous or asynchronous, defined by whether the `SyncRst` or `AsyncRst` ports are set. Moreover, the component has the `In` and `Out` association, describing the

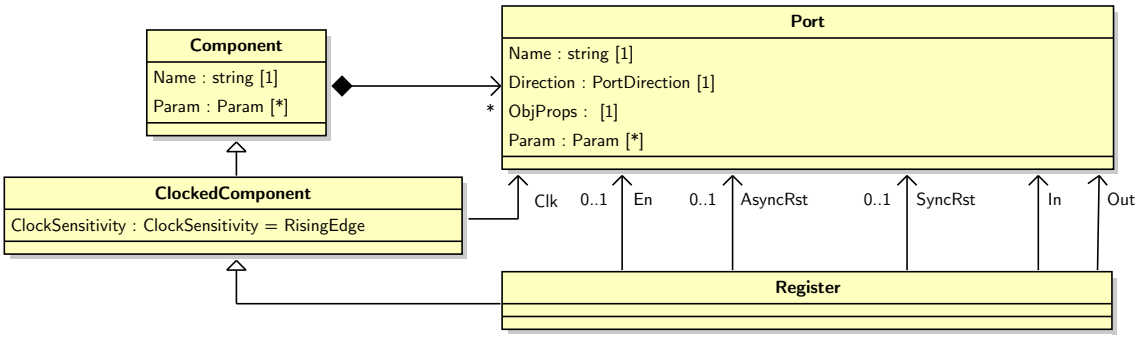

Figure 4.6.: The MetaRTL Register Component

ports which are typically referred to as `D` and `Q` of a FlipFlop. For the MetaRTL Metamodel, the decision was made to name the associations `In` and `Out` for consistency with all other components.

4.3.2.3. Memories

The purpose of the `Memory` component shown in Figure 4.7 is to instantiate memories of all kinds in MetaRTL designs. For this purpose, the Memory component comes with several attributes that can be set to define a certain memory. `ReadLatency` and `WriteLatency` define after how many cycles of the clock `Clk` the data is avaiable. The `ReadUnderWrite` enumeration defines how the Memory behaves when the same address is written and read simultaneously.

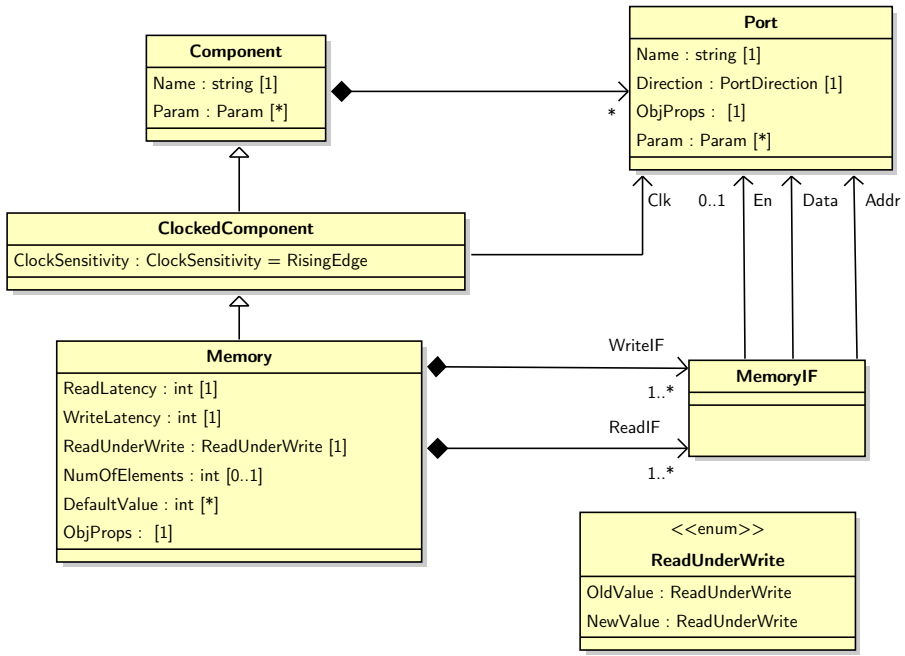

Figure 4.7.: The MetaRTL Memory Component

The `Memory` component also differs from other components introduced so far in how it uses associations to describe the roles of its `Port` instances. As read or write interfaces of memories always require multiple ports, the `MemoryIF` class is defined. Every memory must have at least one `ReadIF` and one `WriteIF`. Each of those interfaces must have an association to two ports, defining it as the `Data` or `Addr` port of the respective memory interface. An optional `En` can be used to disable the writing or reading in a certain cycle. It is optional, and if omitted, the memory will read and write from all its interfaces in every cycle.

4.3.2.4. Primitives

The MetaRTL Metamodel comes with a set of `Primitive` components. Figure 4.8 shows two example Primitives created in the MetaRTL Metamodel. The unary `NOT` operator represents a bitwise not. It is defined for all kinds of numeric interpretations, has exactly one input port (multiplicity of `In` is `1`) and exactly one output port (multiplicity of `Out` is `1`). This is consistent with the definition of the orthogonal operator as a unary operator. The `HWPLUS` operator represents addition with hardware semantics. It is defined on an arbitrary number of arguments, therefore has at least one input port (multiplicity of `In` is `1..*`) and exactly one output port (multiplicity of `Out` is `1`).

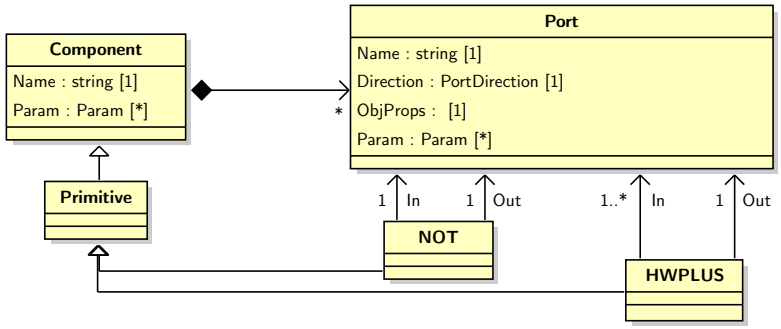

Figure 4.8.: Two sample Expression Operands with their Mapping to Primitives

All `Primitive` components part of MetaRTL are not clocked. The total set of all primitive components can be used to describe arbitrary combinational logic behavior in a structural manner. A detailed overview on MetaRTL Primitives and their auto-creation in the MetaRTL model is provided in a later section.

Together with the Registers and Memories previously introduced and the connectivity defined inside `Structure` components, the components are sufficient to describe any digital design module using MetaRTL.

4.3.2.5. Higher-level design patterns as MetaRTL Components

Although the low-level components introduced so far can already be used on their own to describe any design as a Model of MetaRTL, the Metamodel also contains higher-level artifacts. Examples of these higher-level artifacts are Decoders, Lookup Tables, or Finite State Machines. All three of these examples can certainly be also mapped on a MetaRTL Metamodel using only registers, structures, and primitives. This is however not fully meeting the MetaRTL objective: MetaRTL aims to provide a design-centric model, with hardware semantics and the typical design patterns used for digital design. Decoders, Lookup Tables, and Finite State Machines are important design patterns and an essential part of "designer's thinking". The key problem with this is described as part of the semantics of MetaRTL: mapping these design patterns to lower-level primitives will blur the view and understanding of these patterns and make transformations significantly more difficult.

The remainder of this section describes the `LookupTable` component as an example of a higher-level component embedded into MetaRTL. Figure 4.9 shows the `LookupTable`, a transcoding structure that has a single input `In` and a single output `Out`. It uses a set of nested lookups on certain `KeyBit` positions of the `In` data to determine an output Value. Eventually, every `In` value is mapped to an `Out` value provided by the `ControlVector` attribute of the `TerminalValue` class. It is however also possible to create nested lookups: for a certain lookup `Value`, instead of providing a terminal `ControlVector`, the `NonTerminalValue` class can be used as an additional nested `Lookup`. This lookup then in turn can contain further nested lookups. Eventually, all nested lookups need to end with `TerminalValue` attributes defining what the value at the `Out` of the `LookupTable` component is.

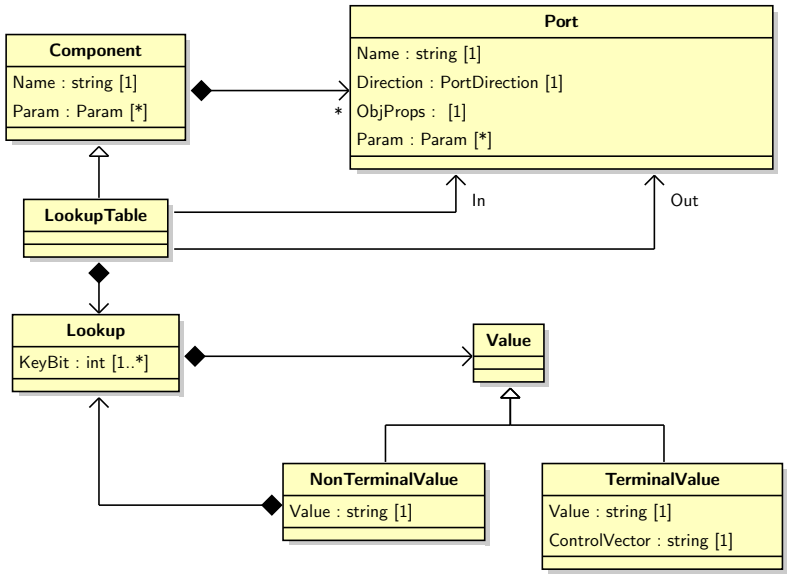

Figure 4.9.: The MetaRTL Lookup Table

4.3.2.6. Compatibility and Hardmacro Components

MetaRTL also contains further specializations of the MetaRTL `Component` needed for the real-world application of the MDA flow. An important example of this is the `Pad` component which can be used to add I/O pad hard macros to MetaRTL models and the `LegacyComponent` to embed any static hardware models provided either by traditional code generators or manually implemented in VHDL or SystemVerilog. These in turn can also cover analog and mixed-signal functionalities such as clock generation and power management. They are not described here in detail as they are not relevant to the conceptual understanding of the MDA method or the Model-of-Design layer.

4.3.3. Sample MoD Layer for Digital Filters

The goal of this section is to further elaborate MetaRTL as Model-of-Design based on the example that was used for the MoT of digital filters in Section 4.2. This section derives an MoD instance for the digital filter that is defined there as $y[n] = 4 \cdot x[n] + 2 \cdot x[n-1] + 1 \cdot x[n-2]$. While the UML instance diagram of the MoT describing this filter in Figure 4.3 is still quite compact and easy to understand, the nature of the MDA flow developed here will inherently lead to more complex models the more layers down the MDA stack.

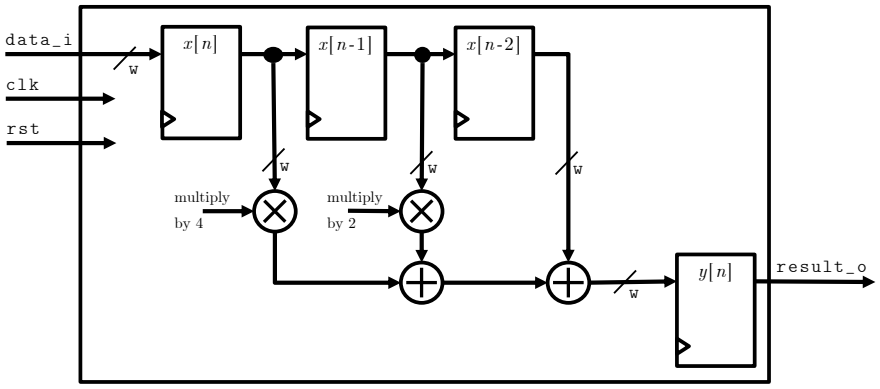

Figure 4.10.: Block diagram of MoD generated from the MoT in Figure 4.3 without Clock and Reset Connectivity

Figure 4.10 shows a block diagram view to illustrate the targeted structural Model-of-Design. The implementation of this filter on a two's complement representation of binary input data of a width `w` can be fairly simple: the mathematical operations of multiplication with 1, 2, and 4 are simple shift operations which in turn can be mapped to simple wires and therefore do not require any dedicated hardware. The addition required to sum up the individual addend can be performed synchronously. It is therefore possible to derive a design that transforms the input into the output in four cycles. For any given cycle n the sum is calculated of the input data two cycles ago ($x[n-2]$) multiplied by 1 (which is simply wiring without any behavior in

hardware), of the input data one cycle ago ($x[n-1]$) multiplied by 2, and of the input data in the current cycle $x[n]$ multiplied by 4.

For different MoT parameters, the complexity and microarchitecture of the filter component can of course differ significantly. This example was chosen specifically to make the Model-of-Design and its visualization as simple as possible. Figure 4.11 shows a UML instance diagram of the Model-of-Design (as an instance of the MetaRTL Metamodel) for the filter from Figure 4.10. The model contains all required components and ports. To ease understanding, it does not represent the real model but a slightly simplified version of it, with similar simplifications as the block diagram in Figure 4.10: the UML instance diagram contains the clock and reset ports but does not contain the clock and reset connectivity. It does further not contain the object properties, i.e. the types of the individual ports, connections, and literals part of the design. Furthermore, it does not contain the `Composition` relations that link the `HWMUL`, `Register` and `HWADD` to their parent `Structure`. A complete visualization of the Model-of-Design with all these attributes would prevent a useful, single-page visualization. Even in its simplified form, it is already clear that the MoD layer differs significantly from the MoT layer. On the MoD layer, is no longer practical to enter models manually, e.g. using a GUI or a defined text format.

To ease the understanding of the UML instance notation of the filter, the blocks in Figure 4.10 and the instances in Figure 4.11 are arranged in the same way. For example, the register `y[n]` that is displayed in the bottom right corner of Figure 4.10 is also in the bottom right corner of Figure 4.11. The figure can therefore be used together with the block diagram to understand how a certain design block diagram translates into an MoD instance.

4.4. The Model-of-View Layer

The Model-of-View layer is used to map the microarchitecture described in a Model-of-Design onto a target platform. In the context of Model Driven Architecture for HW Design Generation, typical targets are HDL code for synthesis or simulation purposes. The Metamodel of the Model-of-View is tailored to the language of the target view.

This means that multiple flavors or styles of the same HDL can be configured using different Model-to-Model transformations between MoD and MoV and mapped to an instance of the same Metamodel-of-View. A Verilog model for ASIC synthesis and a Verilog model targeting an FPGA can be constructed from the same Model-of-Design with two different Model-to-Model transformations. These different transformations of course share significant amounts of code. It is therefore possible to either construct the transformation as one configurable Model-to-Model transformation between MoD and MoV or as two separate Model-to-Model transformations that rely on shared library elements. A third approach supported by the Model Driven Architecture vision is the generation of a Model-of-View and the subsequent application of transformations to this Model-of-View in order to change the coding style to match the target platform.

On the Model-of-View layer, one Metamodel is required per target language. For different target languages (i.e. different HDLs), different Metamodels-of-View are needed. For example,

Figure 4.11.: MetaRTL MoD for Figure 4.10 without Clock and Reset Connectivity (simplified)

in order to generate two different coding styles of Verilog, the same Metamodel can be utilized while a SystemC view will require a different Metamodel. The Metamodel of the Model-of-View layer specifies a targeted view similar to how Extended Backus-Naur Form (EBNF) notations describe the formal grammar of a language. Compared to an EBNF used for parsing, the MoV is more compact, i.e. has fewer classes than the EBNF has rules. Also, the MoV does not have to consider aspects such as left/right recursion, lookup depth and context sensitivity as its only purpose is the definition of a layer for generation, not for parsing. The Metamodel constrains the possible Model-of-View instances so that all legal instances will translate to grammatically correct views.

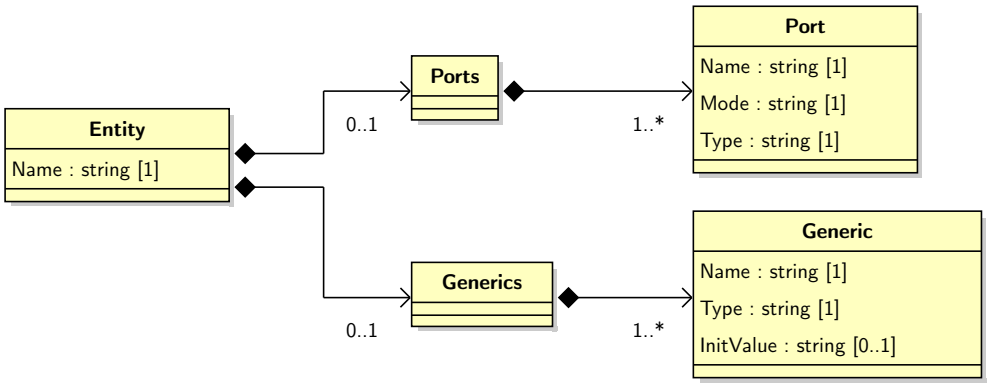

Figure 4.12.: Small Subset of Model-of-View for VHDL

Figure 4.12 contains a small extract from the VHDL Metamodel-of-View which was implemented in the scope of this thesis: this model describes the VHDL `Entity`. In VHDL, entities are the specification of the external interface of a design hierarchy. These are defined by a `Name` attribute (the name of the entity) and the potential `Ports` and `Generics` of the entity. The Metamodel-of-View also contains multiplicities for the defined attributes that make sure that models resulting in invalid RTL would be detected: for example, any defined generic must have a name and a type whereas the initialization value `InitValue` is not mandatory. The complete Model-of-View developed in this thesis contains more classes, attributes, and relations to account for all language aspects required for the generation of RTL code needed for synthesis.

Although there is a straightforward correspondence between the Model-of-View and the generated target views, the MoV-layer of the MDA-inspired approach provides an important abstraction: it allows the developer of the generator to think about the content of the view he wants to generate, without worrying about the need for formatting or indentation (which are not part of any Model-of-View). Formatting and indentation are specified independently of the MoV and utilized to generate the necessary tools to generate the views from the Model-of-View. Consequently, it is also possible to alter these properties without touching the transformation process between any Model-of-Design and the Model-of-View or any transformations performed on an instance of the Model-of-View.

4.5. Orthogonal Metamodels as Abstraction Independent Modeling Artifacts

A key design objective for the definition of the MDA layers for HW design and of all the Metamodels is to provide an MDA flow that makes both the development of new transformations and the application of generators as easy as possible. To simplify the usage of already existing generators for a certain design, it is important to have a clear, understandable, and consistent way of providing Model-of-Things input data. To make generator development as easy as possible, it is important to maximize reuse. A consistent representation of semantically similar artifacts across different Metamodels is essential to achieve this design goal. This applies both for different Metamodels on the same MDA layer and for Metamodels in different layers.

This consistent representation is provided by Orthogonal Metamodel artifacts. The term orthogonal describes that these parts of Metamodels are used in Metamodels across different levels of abstraction (both in Model-of-Things and Model-of-Design) as well as across different Metamodels of the same level of abstraction (in different Metamodels-of-Things).

It is for example necessary to describe object sizes such as the sizes of registers, and the sizes of supported input data for a hardware module on both the Model-of-Things layer and the Model-of-Design layer. The utilization of the same orthogonal descriptions across different Metamodels and layers of abstraction has three key advantages. First, it reduces the development effort for new Metamodels and improves overall code quality and usability. Second, it simplifies the extraction of information necessary for the transformation between different modeling layers. In many cases, objects and expressions can be simply compared and copied between different MoD and MoT instances. Third, the mapping of types and operators to the target view must be implemented only once and can be reused for other targets, e.g. properties [85] which are beyond the scope of this thesis.

The importance of this also shows in today's development landscape: subtle differences in the type systems of different languages, in operator precedence, or the behavior of common mathematical or logical operations are a big hurdle to interoperability. This is apparent in complex IT systems as well as in typical hardware design flows. In the last decade, significant progress has been made in terms of interoperability between software languages [95]. For the task of this thesis, these achievements are however not applicable as they only solve the problem in the domain of software languages. The interoperability the approaches provide does not apply to the domain of digital design with its bit types and values. Moreover, the existing approaches only cover the platform-independent description of types, not the description of Expressions.

For this thesis, a simple, language agnostic, and Metamodel-based type system (referred to as Object Properties) and operator system has been developed. The operators and types defined here are intended for use in Metamodels on all MDA layers wherever Metamodels contain elements describing objects and behavior.

Comparing this to a code generation framework that is based on Domain-Specific Languages (DSLs), Embedded DSLs, or Libraries implemented in different languages shows how important this benefit is. The simple transfer of a simple boolean logic expression evaluating from Python

into C will require the handling of countless corner cases: Python uses, depending on the language version, fixed-precision floating point arithmetic or arbitrary precision arithmetic. The latter is especially difficult to map to lower-level languages and environments. Mismatches such as this alone can become a limiting factor to automation and reuse – the MDA-based approach presented here avoids these issues entirely with its Orthogonal Metamodel.

4.5.1. Orthogonal Object Properties

The Object Properties Metamodel follows the idea of describing data and interpretation using the underlying semantics of hardware wires. Therefore, it supports the specification of any kind of bundles of wires together with some optional hardware properties such as signed interpretation and Endianness. Object Properties can be named to ease the readability of the generated targets and to simplify referencing types in specifications. Type compatibility is however not dependent on type names or other hardware properties. Instead, type consistency is only dependent on the size (i.e. the number of bits or hardware wires) of any two types.

Figure 4.13 shows a UML representation of the Object Properties Metamodel.

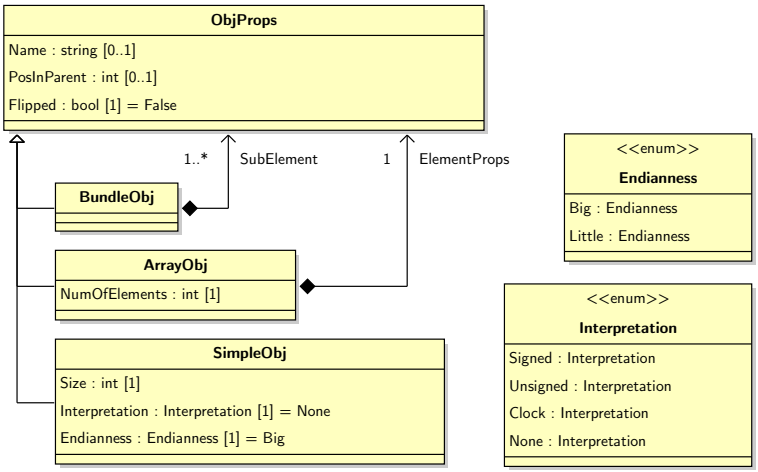

Figure 4.13.: Orthogonal `ObjectProperties` Metamodel

The `ObjProps` Metamodel permits the following types:

- A non-complex, single `SimpleObj`. This describes a standard wire or wire vector without any hierarchy. It is defined by a `Size` attribute describing the number of bits of the object. Moreover, an `Interpretation` determines whether it contains a Signed or Unsigned value or none of these properties. An `Endianness` describes how multi-byte artifacts are serialized.

- `ArrayObj`, which represents a combination of multiple objects with the same properties. This describes a list of a defined length of other objects. The `ElementProps` composition

contains all properties of the individual elements contained in the array, the `ArrayObj` itself just describes how many elements of `ElementProps` there are.

- `BundleObj`, which represents a combination of multiple objects with different properties. A bundle describes a set of different `ObjProps` combined into one bundle. Which `ObjProps` are combined is described by the `SubElement` composition. This combination is comparable to the SystemVerilog record or structs in the C programming language families.

A key element all `ObjProps` have in common is the `Flipped` attribute. It describes the relative direction of the object compared to a potential overlying direction defined for example by a Port of a module. If an `ObjProps` instance is used to describe the properties of a MetaRTL port and the `Flipped` attribute is set to false, the direction of the wires equals the direction of the port. If any `Flipped` attribute is set to true, the direction of the wires described by this attribute will be the opposite of the direction described for the port. The flipped attribute is used to describe subsets of SystemVerilog interfaces based on the Metamodel.

4.5.1.1. Application Example

It is easy to picture a scenario where consistent use of the `ObjProps` defined here is beneficial. It is for example possible to demonstrate the benefits using the `FIRFilter` Metamodel-of-Things defined in Figure 4.2.

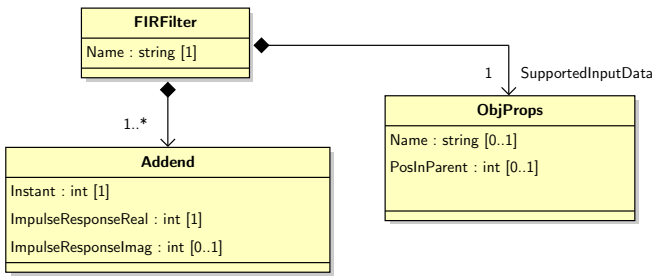

Figure 4.14.: Metamodel-of-Things for FIR Filters using the Orthogonal Object Properties

Figure 4.14 shows how an additional composition named `SupportedInputData` uses `ObjProps` to describe the type and size of data the generated design has to be able to process. The sample MoT from Figure 4.3 can now in turn be extended, resulting in Figure 4.15. The instance of `SimpleObj` defines that the `FIRFilter` has to be able to handle unsigned inputs of up to 64 bits in width. This instance of `SimpleObj` can be transferred without any modification to the downstream Model-of-Design. As a result, the Model-to-Model transformation between MoT and MoD is significantly simplified. This is not illustrated in detail- as the Model-of-Design is even without the object properties close to a size where a meaningful visualization is no longer possible.

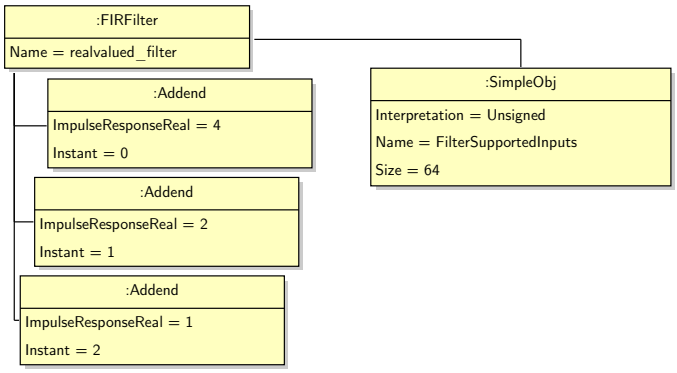

Figure 4.15.: Model-of-Things for FIR Filters using the Orthogonal Object Properties

4.5.2. Orthogonal Operators for Expressions, Dataflow and Structural RTL

The operators developed in this thesis are platform-independent and cover both hardware and software semantics. The main design goal for the set of defined operators is to make adoption as easy as possible: it should be very simple to understand and any potential confusion about operator semantics should be avoided. Two key design decisions are important to achieve this:

1. Symbols for operators are avoided entirely. Instead, operators are named by text-based mnemonics. This solves two issues:

   a) The amount of symbols available for operators is limited and the support for both hardware and software semantics requires a larger set of operators.

   b) Many symbol-based operators are already used by existing languages with predefined semantics that differ from language to language. Depending on the background of the user of the MDA framework, different semantics of operators leads to potential confusion and errors.

2. The presented approach does not rely on Infix Notation where the operator is placed in between the operands (the example "$a+b$" uses Infix Notation). Instead, a Prefix or Polish Notation is used where the operator precedes the arguments (the example "$+(a, b)$" uses Polish Notation). The suggested approach further relies on an expression syntax where the mnemonic operator precedes the arguments which are in turn comma-separated and enclosed in brackets. This comes with two main benefits:

   a) In almost all programming languages, the operators can be mapped to function or method calls. It is therefore easily possible to implement the operators and expression syntax as an embedded DSL in any language without dealing with the complexities of parsing.

   b) The question of operator precedence does not arise. There is no need for a defined operator priority or a counterpart to brackets for modifying the default operator

priority. This is particularly important as operator precedence is often different for the same operators in different languages, another source of potential confusion and errors.

For each operator, the size of the output depends on the size of the inputs. The relationship of input object properties to output object properties is generically defined and independent of the context and semantics of where the operators are used. Moreover, there is no size limit as such.

The operators defined and implemented are listed in Table 4.1. The different semantics between hardware and software is accounted for by providing different operators for example for multiplication (HWMUL, CMULT) and for additional (HWPLUS, CPLUS). Examples of the availability of dedicated hardware operators are the bitwise operations (e.g. BOR, BAND, ...) and reduction operations (e.g. ROR, RAND). The differences between these operators can be easily illustrated with an example: the result of a CPLUS operator has the maximum size of its inputs (an addition of two 32 bit values will result in a 32 bit wide result value). The result of a HWPLUS operator on the same inputs will be two bits wider to represent the carry and overflow bits as well.

This definition of operators and its characteristics comes as a model of its own Metamodel. This model can in turn be used to define multiple further Metamodels. In the scope of this thesis, Expression, Dataflow, and Structural representations are generated using the Expression Metamodel.

4.5.2.1. Usage in Expression Metamodel

The Expression Metamodel shown in Figure 4.16 defines a tree of operators, with both classical operators from programming and specific hardware operators. Figure 4.16 shows the Expression Metamodel with a subset of the Operators defined in Table 4.1. The diagram shows that the expressions form an expression tree defining the expression hierarchy. Leaf nodes of this tree can be binary or numeric literals in any combination as well as references to objects with properties defined by the type model or variables which act as input to expressions.

4.5.2.2. Usage in Dataflow Metamodel

Figure 4.17 shows the Dataflow Metamodel with a subset of the Operators defined in Table 4.1. The Expression Model described in the previous section is a special case of a Dataflow Model. This means every expression model can be transformed automatically into a dataflow model, while the opposite is not possible. It is different from the Expression Model as any operator output can act as input to multiple dataflow operators. The Dataflow Metamodel does therefore not necessarily define a tree structure but a directed acyclic graph.

Operator	Description	Operands
NOT	bitwise not, defined for all numeric interpretations	1
CABS	computational arithmetic absolute value	1
CUMINUS	computational arithmetic unary minus	1
SLICE	select slice of <op1> from upper index <op2> to lower index <op3>	3
HEAD, TAIL	select <op2> bits from upper/lower bound of <op1>	2
INDEX	select individual bit number <op2> from <op1>	2
REVERSE	reverse bits left-to-right turned into right-to-left	1
ROR, RAND, RNAND, RNOR, RXOR, RXNOR	reducing boolean operators	1..*
BOR, BAND, BNAND, BNOR, BXOR, BXNOR	bitwise operators	2..*
LOR, LAND, LNAND, LNOR, LXOR, LXNOR	logical operators	2..*
CPLUS, CMINUS, CMULT	computational addition, subtraction, multiplication	2..*
CDIV, CMOD, CREM	computational division, modulus, remainder	2
SIGNEDCAST, UNSIGNEDCAST	signed and unsigned cast (does not modify the value, change bit width, etc.)	1
UNSIGNEDCONV	unsigned conversion (does not modify value but cuts the highest bit if the input was signed)	1
SIGNEDCONV	signed conversion (does not modify value but changes bit width)	1
HWMUL	multiplication with hardware semantics	2..*
HWPLUS	addition with hardware semantics	2..*
RSL	right shift logical (fill with zeros on the left)	2
RSA	right shift arithmetic (fill with sign bit on the left)	2
LS	left shift (fill with zeros on the right)	2
CONCAT	concatenation	2..*
MUX	if <cond> then <trueexpr> else <falseexpr>	3
ISNEG, ISPOS	return 1 if negative/positive value provided, return 0 if positive/negative value provided	1
LT, LTEQ, GT, GTEQ, EQ, NEQ	logical comparison operators	2

Table 4.1.: MDA Operators

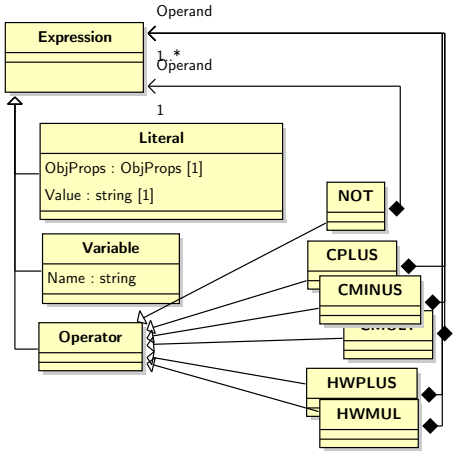

Figure 4.16.: Orthogonal Expression Metamodel with a subset of the available Operators

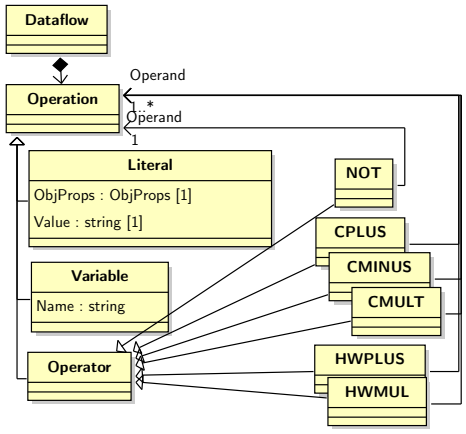

Figure 4.17.: Orthogonal Dataflow Metamodel with a subset of the available Operators

The main difference between Dataflow and Expression models is that a Dataflow model possesses no tree hierarchy. Instead, a Dataflow model is directly composed of a list of all Operations. These operations may be of the same type as the Expressions of the Expression Metamodel: they can be Literals, Variables, or Operators. For all operators, the Operands are no longer compositions of the Operation. Instead, the operation references its operands by means of associations.

4.5.2.3. Usage as MetaRTL Primitive Components

As described in Section 4.3.2.4, MetaRTL comes with a set of `Primitive` components. These components are automatically generated from the defined operators and are therefore consistent with the orthogonal Expression Metamodel and Dataflow Metamodel described above. For use as MetaRTL components, the definitions of the operators with their operands are mapped onto the semantics of the `Component` with its `Ports`.

For every operand in the expression language, a simple, non-clocked component is created in the MetaRTL Metamodel. According to the definition of the orthogonal expression operators, all operators have one result or return value and one to many inputs or operands.

The `Primitive` components part of the MetaRTL Metamodel are automatically added to the Metamodel from the same single source as the Expressions in MoT instances. The transformation into a Component and the MoD specific `Port` properties together with the ability to name individual components and to connect them in hardware semantics is added during the automated Metamodel creation process. This guarantees that the expression semantics of the MetaRTL Model-of-Design is always compatible as long as both Metamodels use the same orthogonal Metamodels.

5. Development of a Model Driven Architecture Framework

Chapter 4 describes an important aspect of a successful application of Model Driven Architecture to the hardware domain: it defines the right levels of abstraction and the right Metamodels for a productive MDA-based generation framework. While it is an important contribution of this thesis and an important prerequisite for a successful application of Model Driven Architecture, it is only one of the key components to achieving the goal of this thesis (see Section 2.5).

Figure 5.1 shows important components that have been omitted from previous visualizations of the approach (e.g. Figure 4.1) and that have so far not been discussed in this work: the *Readers* needed to access Specification information, the Model-to-Model transformations *Template-of-Design* and *Template-of-View* as well as the *Unparse View Generation* mechanism. All these components are transformations of some kind: Readers can be seen as transformations from serialized data into models part of a Metamodeling framework, and view generation is a Model-to-View transformation. The Model-to-Model transformations *Template-of-Design* and *Template-of-View* are in turn what is seen by many as the "heart and soul of any model-driven development approach" [45, 26].

The properties of these components are inherently tied to the utilized framework and development ecosystem. A theoretical, language-agnostic assessment only based on the fundamentals of Metamodeling is therefore not meaningful. The development of the Model Driven Architecture Framework is where the language-agnostic world for Metamodeling meets a specific implementation and its tools and languages. This chapter details the implementation and the design decisions made as part of the development of the Model Driven Architecture framework provided by this thesis.

This chapter first introduces Infineon's Proprietary Metamodeling framework and alternatives to it and explains why the existing proprietary framework is ideally suited as the

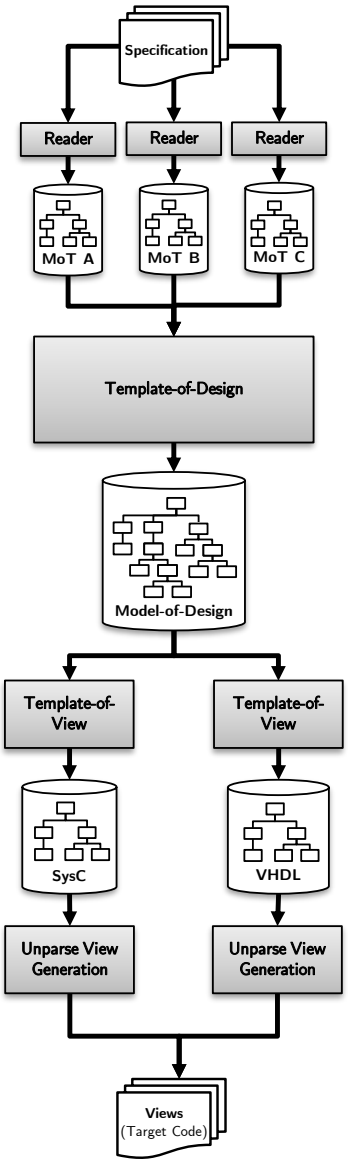

Figure 5.1.: Readers, Model-to-Model Transformations and View Generation

foundation for the Model Driven Architecture flow. To understand these design decisions, the properties of the Infineon Metamodeling framework are put in context to the high-level goals of this thesis and in context to the engineering environment in which the MDA approach is introduced.

The adaptation of the Infineon Metamodeling Framework for Model Driven Architecture is shown next. In this context, it is also described how Metamodel-based automated generation of tools is related to an efficient MDA flow.

The next paragraph describes the approach taken to Model-to-Model transformations and how the Metamodeling Framework has been extended to enable efficient development of the core component of the MDA framework, the *Template-of-Design* transformation from MoT to MoD layer.

Eventually, this chapter presents a key contribution that further helps to automate the generation of as many framework components as possible: the fully automated *Unparse View Generation* enabled by the *View Language Description (VLD)* approach pioneered by this thesis.

5.1. The Infineon Metamodeling Framework

This section describes why Infineon's Proprietary Metamodeling Framework *Metagen* is used as the foundation for the MDA framework developed in this thesis. It also compares Infineon's approach to the Eclipse Modeling Framework to justify the decision to rely on Infineon's proprietary framework for this thesis.

Infineon relies on a proprietary Metamodeling Framework as a foundation for its Metamodeling-based code generation. It provides an implementation of the Metamodeling concept as described in Section 2.3.2 and its usage at Infineon for Metamodeling based code generation as described in Section 2.3.3.

The Metagen framework is well-established at Infineon and has demonstrated the capability to deliver significant increases in productivity for overall development tasks. Ecker et al. [63] show a factor 20 in special design tasks and up to a factor of three in design implementation from specification freeze to tape-out through the use of this framework.

5.1.1. Programming Language

A key strength of this framework is the fact that it is implemented in the Python programming language. Python is ideally suited for a Model Driven Architecture framework for Hardware Design for the following reasons:

- Python's syntax is simple and bears similarities to natural languages, leading to a particularly low entry barrier for beginners and users outside of the field of software engineering.

- Python is an interpreted language. Users can therefore run their code without worrying about compilation and development environments - this is even possible in interactive shells.

- Python is the de facto standard scripting language. It is the primary language taught in engineering courses outside the domain of software engineering. It has the largest user base of all programming languages, therefore providing an infinite amount of online support, tutorial, and training resources.

- Python comes with a large set of libraries that are useful to the field of IC design. These libraries include data analysis, machine learning, and scientific computing frameworks. Moreover, it is a popular language for developing simple web applications – thus providing its users with a set of useful template engine libraries.

- Python is a high-level language that abstracts the details of the underlying platform – users of the language do not need to worry about low-level programming details such as memory management

- Python offers strong support for object-oriented programming, functional programming, and introspection. These are important patterns required to support Metamodeling concepts.

An important drawback of Python is its dynamic typing. It is generally accepted that dynamically typed languages are not suited to build complex enterprise software systems with many actors. When comparing dynamically typed languages to their statically typed siblings of the same level of abstraction, the developer productivity in the language itself is consistently lower. This is caused by several main factors.

- State-of-the-art Integrated Development Environments (IDEs) for dynamically typed languages cannot provide the same level of assistance and design analysis as for their statically typed siblings. Many mistakes that are introduced during the development (e.g. typos in member names, incorrect function names, or signatures) can only be detected at run-time because of the missing and ambiguous type information and the languages' capabilities to modify types and objects at run-time [29, 59, 62].

- The dynamic nature allows for violations of software engineering best practices that are impossible, hard to achieve, or easy to spot in statically typed languages. Anti-patterns such as monkey-typing, variables that can at run-time contain members of different types are common in Python code-bases. The technical debt that comes with this is difficult to prevent (it is challenging to spot these anti-patterns when they occur) and hard to fix (refactoring dynamically typed code is not generally possible as no automated way to analyze dependencies exists) [29, 59, 62].

- The missing type information in code makes it difficult to read code written by other developers. In mature software projects, reading, understanding, and modifying existing code bases thus becomes more time-consuming. Dynamically typed languages are therefore especially poorly suited for environments with high employee turnover [29, 59, 62].

- Python is the modern programming language with the lowest overall execution performance. This is partially caused by its dynamic nature. A main part of it however has to be attributed to a lack of investment in the performance of the language. The main reason for this missing investment is the fact that Python as a scripting language is not well suited and therefore not widely used to deliver complex software projects. In consequence, there are few actors who would benefit from the performance improvements in the language. It is important to note that recently, larger actors in the domain of software engineering are contributing to the improved performance of Python and potential future contributions will also benefit existing code.

Despite these drawbacks, Infineon's Python-based Proprietary Metamodeling Framework is best suited to establish a Model Driven Architecture approach for Digital Hardware Generation. The key arguments for this decision are:

- A low barrier to adoption is crucial for the approach. The success of Metamodel-based code generation at Infineon has only been made possible because the Infineon Metamodeling Framework allowed domain experts to contribute efficiently to generators. The same contribution will also be required for the Model Driven Architecture framework developed in this thesis. This thesis clearly demonstrates the benefits of an MDA approach with Python, the language is therefore good enough to achieve the objective of MDA.

- Many of the performance limitations can be overcome and recent developments in the Python ecosystem provide proof that the performance problem is one that will be alleviated over time – without further investment into the development of the MDA framework.

- Metamodeling as such is a language-independent concept. When it comes to the Infineon Metamodeling Framework, implementations of the core components with support for all Metamodels of the frameworks' Meta-Metamodel also exists for languages other than Python. In particular, the Infineon Metamodeling Framework supports an API generator for C++ for use cases that require extreme performance. Moreover, an API generator for C# is available for use cases that require enterprise-grade scalability and high developer productivity. It is therefore already possible to develop individual elements of the framework in different languages – a capability that can be utilized to resolve potential critical performance bottlenecks that might show up in larger-scale applications of the MDA-based generation approach.

5.1.2. Alternative Metamodeling Frameworks

The Eclipse Modeling Framework (EMF) is the one noteworthy alternative in the landscape of Metamodeling. It is suited for use in an MDA approach as it supports all key requirements and is available as open source. An assessment of the framework (see [49]) shows that it is even more powerful than Infineon's internal proprietary framework. This however comes at the cost of significant complexity.

The Java-based framework depends on infrastructure components from the Eclipse ecosystem and was designed with the best practices of Java software development in mind. While this makes it well-suited as a foundation for a Model Driven Architecture approach to software engineering, it also makes it very hard to learn and adopt for hardware engineers. The Infineon Metamodeling Framework in contrast is targeted at users which are not professional software engineers. The Infineon Metamodeling Framework thus has a significantly lower barrier to adoption – both for the Python-based implementation and the C# implementation of the core components.

An important requirement at the center of the Model Driven Architecture approach is the ability to apply and scale the methodology in a commercial hardware development environment. It thus has to be possible to develop generators for hardware designers who have knowledge of scripting and code generation but are not trained as software engineers. This is a fundamentally different goal than the one underlying EMF and it is therefore clear that EMF cannot serve as a viable alternative to Infineon's in-house framework for the use case at hand.

5.1.3. Metamodeling Framework Architecture

The Metagen Metamodeling framework uses a subset of UML class diagrams, extended with some features from XML Schema to specify Metamodels. In essence, Metamodels specify the definition of objects, their attributes, and various kinds of relations between them.

The Metamodel can be captured in a textural and graphical way. For this purpose, the Infineon Metamodeling Framework utilizes a UML modeling tool called BoUML. The UML model defined in this tool can then be read into the Infineon Metamodeling Framework. This reading of the Metamodel is based on the same flow as the reading of models, as the Metamodel is described as an instance of the Meta-Metamodel of the framework. This Meta-Metamodel can also be written in a textual format or be defined exclusively with the textual format, without using the graphical UML tool.

The architecture of the Infineon Metamodeling Framework is heavily centered around code generation. Figure 5.2 shows a high-level view of the architecture. For any given Metamodel, the Infineon Metamodeling Framework generates a set of tools and framework components that then make up the modeling environment for this Metamodel. These autogenerated tools are shown in the dashed box "Modeling Framework Autogenerated from Metamodel" in the

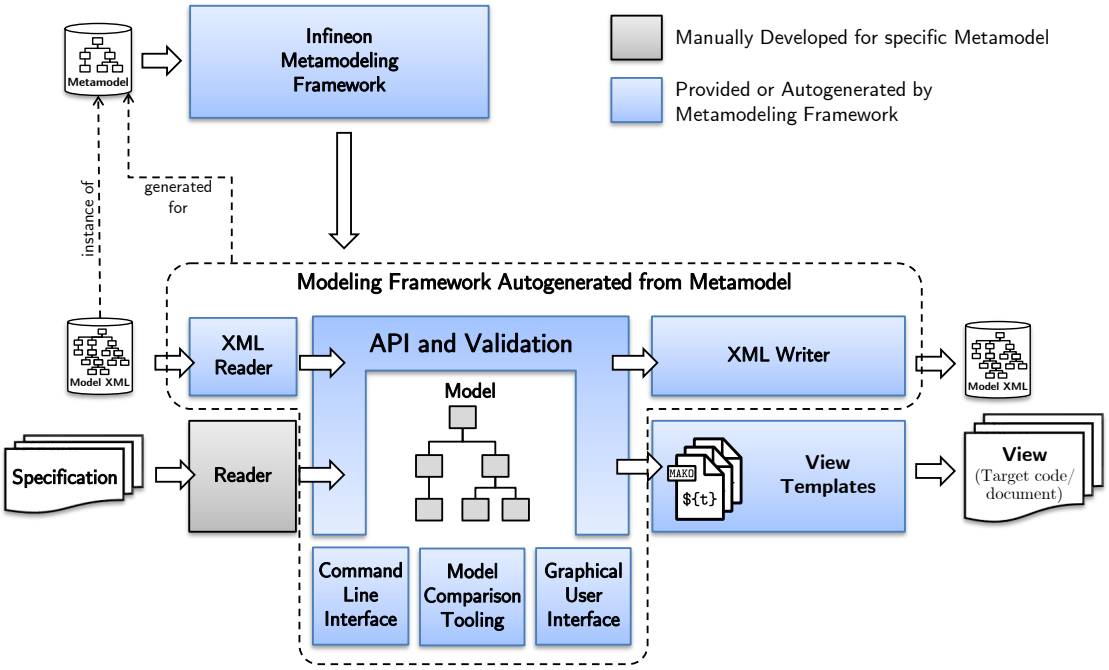

Figure 5.2.: Abstraction Layers in MDA for Hardware Generation – Models and Metamodels

figure.

The high-level structure of the generated component is made up of three main parts. First, a set of *Readers* that load models from sources external to the Metamodeling framework. The second part is the core component of the generated, Metamodel-specific framework: the API wrapping the data structure that contains the model. After the reading step is completed, it stores the model inside the framework. This part provides getter and setter features and validation features that ensure that the model adheres to the constraints defined by the Metamodel. The third part is a set of *Writers* which consumes a model stored in the API and stores the models to destinations external to the Metamodeling framework.

In addition to the elements described above, the Metamodeling framework provides several important infrastructure components:

- A Python-based template engine using the Mako template library is part of the framework. It is therefore a standard approach to use Mako-format templates for view generation. These templates are so far the standard approach to turn the model information into views for consumption by tooling. Several standard templates, for example for visualization of the model, are autogenerated by the Metamodeling framework. The standard use case is however to manually develop Metamodel-specific templates.

- For every Metamodel, a set of command line tools is generated to invoke the framework. Different flavors of these command line tools can be used to perform Model-to-Model

transformations, reading and writing to and from various different formats, and call the template engine to generate target views. The generated command line tools also support joint handling of multiple models of the same or different Metamodels.

- A plugin mechanism to analyze, modify and transform models after they are read and before the writers are called.

- A Graphical User Interface (GUI) for the Metamodel is generated that allows convenient input and utilization of the validation features provided by the API. This GUI is tailored to the Metamodel and provides input interfaces specifically for the objects, attributes, and relations defined in the Metamodel. This provides a no-cost and no-overhead way to interface any Metamodel-based generator.

- Various other Metamodel-generated toolings. Most notably, a Model Comparison Tool is available to identify and visualize differences between the model elements. Due to the utilization of Metaprogramming in the generation of these additional Metamodel-specific tooling, the environment for any existing Metamodel will also benefit from further features added to the Metamodeling framework at a later point in time.

The benefit and productivity increase provided by the generation of a custom-tailored framework from the Metamodel can be easily explained with a simple phrase: the Infineon Metamodeling framework is to a significant extent *generating the generator*.

It is important to note that the Infineon Metamodeling Framework itself is built on the same internal structure as the modeling framework that it generates: a set of different readers can read the Metamodel descriptions into an API built for the Meta-Metamodel. Different readers and writers are available to support various different input formats (e.g. the text-based format described above). It is also possible to modify, preprocess, merge, or completely autogenerate Metamodels using this framework. Section 5.4 discusses an example of such an autogenerated Metamodel based on the View Language Description approach developed as part of this thesis.

The uniform structure of the modeling flow also makes it easy to understand and use the Metamodels. This ease of use and simplicity is especially helpful when applying the framework to build a Model Driven Architecture inspired multi-step generation flow: the different levels of detail resulting from different levels of abstraction from MoT to MoV can be easily handled with the same framework schema. Descriptions can be made in a uniform way, transformations are less painful and there are fewer inconsistencies.

5.1.4. Development Environment: Package, Dependency and Environment Management

Infineon Metamodeling framework does not just come as a library or generator that can be utilized for development. Instead, it is embedded into a state-of-the-art software development framework and flow. Such a flow provides three main components: Package Management,

Dependency Management, and Environment Management. The following paragraphs explain the role these components play and why they are required for a hardware generation flow [88].

Package Management describes the process of packaging and distributing code as software components or libraries. Package Management Systems provide the ability to package software components, distribute and provide them to a centralized repository location, as well as to install, uninstall and update them in a certain setup. Package Management is however not only required for software development. Hardware Design comes with the same requirements, as complex ICs need to make use of packable, reusable sub-components, so-called IP components. The similarity of these IP components (i.e. modules that are instantiated in the integrated circuit) to software libraries (i.e. modules that are utilized by a larger software product) is apparent.

Dependency Management describes the process of defining dependencies between different packages. Dependency management systems in the context of software provide the ability to define which version or range of versions of a certain dependent package is required to compile or run a piece of software. Dependency Management Systems are required to ensure that the required software components for a project are compatible and to ensure that a reproducible combination of packages is used to build and deploy a given piece of software. The same also applies to Hardware Design: Complex ICs need to be assembled from the right, well-known, and reproducible IP components based on a predefined bill of material.

Environment Management describes the process of providing one or more separate, independent environments for defined project tasks. Environment Management will ensure a certain environment contains only the packages described by the Dependency Management System and format. It will further ensure the compatibility of all dependencies and provide a reproducible report containing all utilized packages. Environment Management is important to prevent the accidental use of incorrect, outdated, or experimental artifacts in both software and hardware development. Moreover, a development flow needs to support different environments where different states or different projects with different dependencies can be developed independently of one another.

Package, Dependency, and Environment Management are well-established components of software development flows, often even tightly integrated into the build system and development environment of a certain programming language. The field of IC design has come up with its own solutions for Package, Dependency, and Environment Management. These solutions are typically either proprietary commercial systems or solutions that originate as in-house developments of semiconductor companies.

The Python language itself lacks a built-in Package, Dependency, and Environment Management System. The Development Flow however adds these components to the Metamodeling Framework and the Python ecosystem: the flow relies on Conda [88], an open-source, cross-

platform package management solution. For the application of Model Driven Architecture to Hardware Design, this can be fully utilized. The reason for this is simple: the MDA layers, Metamodels, and Model-to-Model transformations that are used to generate the hardware are inherently just Python software libraries. They therefore can be entirely managed by the software development flow integrated into Infineon's Metamodeling environment. The need for a separate, custom Package, Dependency, and Environment Management tailored for hardware is eliminated.

This emphasizes a key benefit of the software and Metamodeling-based approach to developing generators instead of developing instances of the design: the powerful ecosystems software engineering has developed can be utilized for the hardware domain.

5.2. Model Driven Architecture based on the Infineon Metamodeling Framework

Section 5.1 describes the Infineon Metamodeling Framework with its language, architecture, and development environment. It puts particular emphasis on how the concept of *generating the generator* leads to high productivity and quality during the development of Metamodel-based code generators.

This concept is very valuable in order to achieve the high-level goal of this thesis: to provide even more powerful and efficient Metamodeling-based automated code generation. The following shows how the Infineon Metamodeling Framework is utilized as the foundation for the development of the Model Driven Architecture framework and how the concept of *generating the generator* can be applied to the majority of MDA framework components. For this purpose, this thesis provides a refinement of Figure 5.1 from the beginning of this chapter: Figure 5.3 reuses the high-level view on the dataflow and transformations of the MDA framework and adds all software components of the MDA framework this thesis derives. The color coding indicates which artifacts are automatically generated (solid blue) or pre-developed as part of this thesis (dotted blue) and which artifacts are to be provided by developers of MDA-based code generators or users of the generators (yellow).

Without further diving into the details of this high-level representation, a key aspect becomes apparent: The approach implemented in this thesis bridges a high level of abstraction. Despite this high level of abstraction that is bridged, the amount of work that developers and users of generators need to invest is limited to a small area because Metamodeling-based automation, code generation, and library elements provided by the research work are able to abstract away significant parts of the complexity of the generation task.

The following paragraph discusses the figure in top-to-bottom order. At the top, the figure shows the specification provided by users of the generation flow. This specification can be automatically read from XML files with the readers the Metamodeling framework generates for a defined Metamodel. Alternatively, the users can utilize the autogenerated GUI provided by

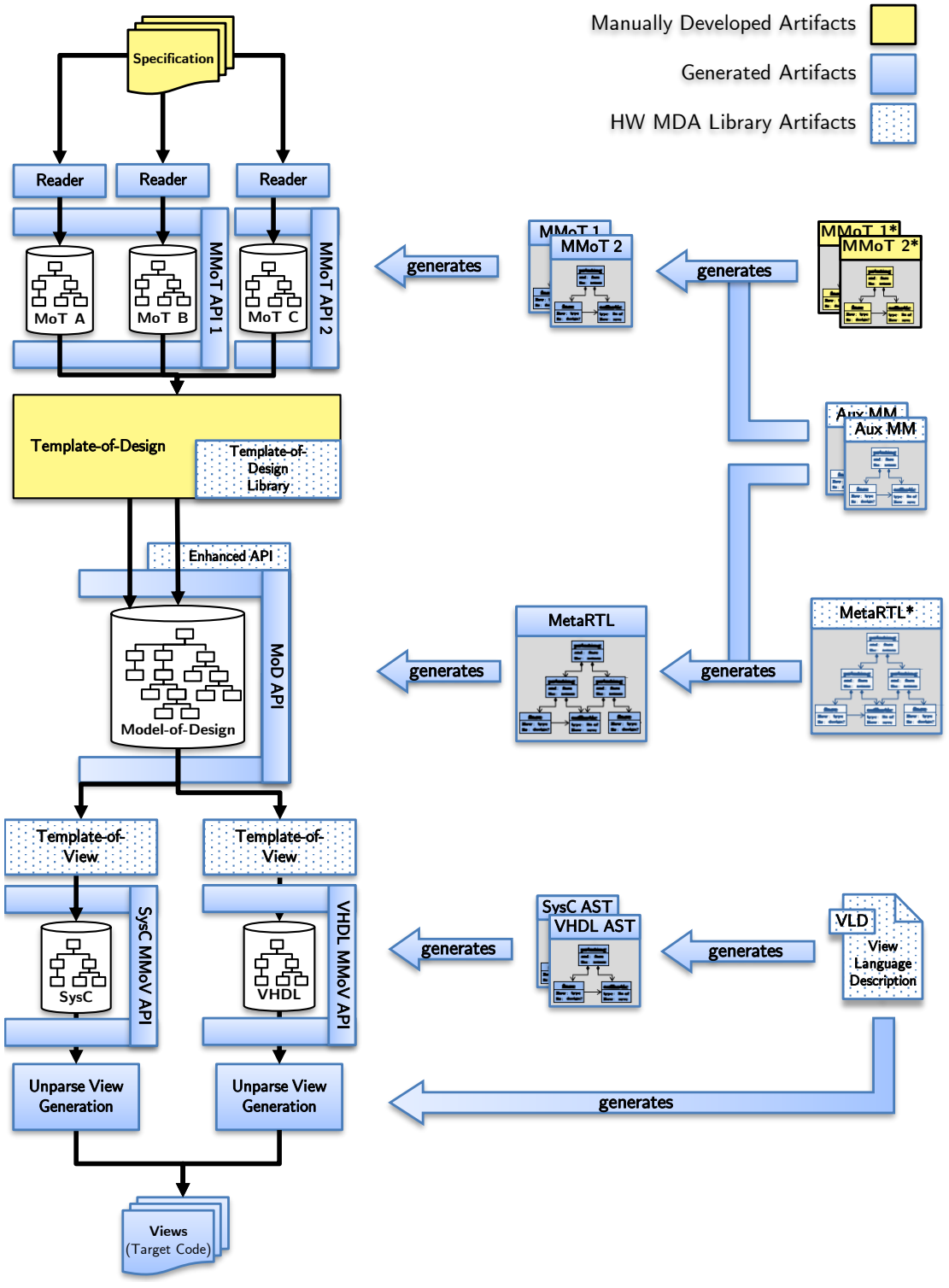

Figure 5.3.: Abstraction Layers in MDA for Hardware Generation – Models and Metamodels

the framework as well as a set of other data analysis tools to provide the model data. All that is required for this automation is the definition of the Metamodel-of-Things which has to be provided by the developer of the MDA-based code generator (MMoT 1* and MMoT 2* in the figure). It is important to note in this context that the construction of these Metamodels is not a coding task but a modeling task.

To further simplify this modeling task, Metamodel-of-Things Metamodels do not necessarily need to be fully defined by the developer. As described in Section 4.5, Orthogonal Metamodels (abbreviated as Aux MM in the figure) reduce the effort. These Orthogonal Metamodels act as placeholders for modeling aspects common to multiple layers of the framework. A Metamodel-merge mechanism can generate Metamodels containing these orthogonal elements (MMoT 1 and MMoT 2) from developer-defined Metamodels which only reference them (MMoT 1* and MMoT 2*). The same is also true for the Metamodel-of-Design MetaRTL. A version of MetaRTL* that does not contain the Type and Expression Orthogonal Metamodels is modified by a merge process which then creates MetaRTL. This process ensures consistency between different Metamodel-of-Things instances and between the Metamodels-of-Things (MMoTs) and the Metamodel-of-Design (here: MetaRTL).

The Metamodels-of-Things and the Metamodel-of-Design are then used to autogenerate the APIs of the MoD and MoT layers. These APIs define the interfaces used by the first and most important Model-to-Model transformation: the *Template-of-Design (ToD)*. The Template-of-Design is the key component to which a developer of generators using the Model Driven Architecture framework is exposed. Section 5.3 describes the mechanism of Model-to-Model transformations that is introduced by this thesis and also details how library elements (Template-of-Design Library) and an Enhanced API developed as part of this thesis contribute to the overall goal of making the development of generators highly productive.

After the Model-of-Design is constructed by the Template-of-Design Model-to-Model transformation, the third layer of the Metamodeling approach is reached. The Template-of-View is a Model-to-Model transformation utilizing a Python-based approach similar to the Template-of-Design approach utilized for the MoD construction. The development of a Template-of-View for a different language or the adaptation of a Template-of-View for a different generation flavor is typically a one-off task. As part of this thesis work, a Template-of-View for HDL generation in the VHDL language was developed. Further internships and PhD theses have utilized the MDA approach described here and applied it to different target languages (e.g. (System-) Verilog HDL and C-based firmware and embedded software as well as SVA, and ITL to generate properties for formal verification purposes). The Template-of-Views are visualized in dotted blue as any Template-of-View that is generated once can be reused for all existing and future generators (i.e. for all Templates-of-Design transformations and MMoVs).

Section 4.4 describes the MoV, its level of abstraction, and how it benefits the MDA approach. In the implementation of this approach, it has been identified that it is cumbersome to manually define an AST-based Metamodel for the MoV layer. Moreover, the generation of views and different flavors of views from this AST-based Metamodel is tedious to implement. This challenge led to the complete automation of the generation of the Metamodels-of-View and of the view generation (*Unparse View Generation*) from the MoV. This approach is provided by a *View*

Language Description format similar to EBNF that was developed in the scope of this work. Section 5.4 describes this approach in detail. For this section, it is sufficient to note that the entire MoV including the MMoV and MoV-to-View transformation is automatically generated from a configurable library component called VLD from a more abstract description.

5.3. Model-to-Model Transformations

The use of very abstract models and their gradual refinement through Model-to-Model transformations is one of the key concepts at the heart of the Model Driven Architecture Vision. The development of these Model-to-Model transformations is therefore an essential aspect of applying Model Driven Architecture to HW design. Figure 5.3 highlights the importance of easy-to-develop transformations: the Template-of-Design and Template-of-View are the MDA artifacts that need to be manually developed (a Template-of-Design needs to be developed for every design generator and a Template-of-View needs to be developed for every targeted view language).

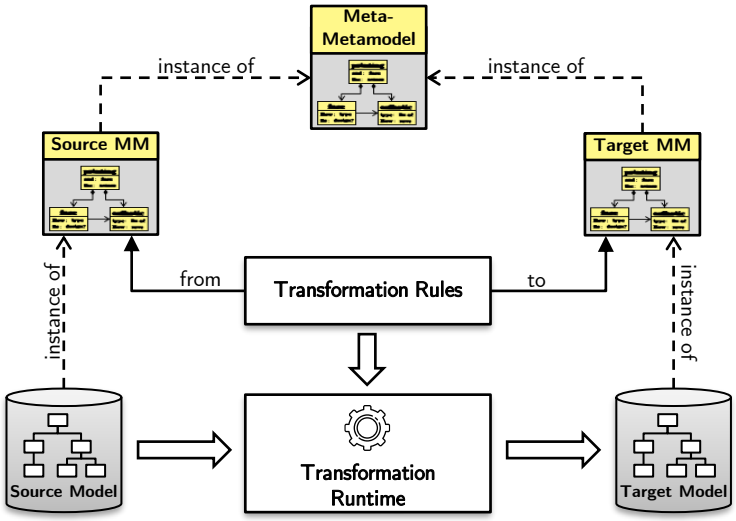

Figure 5.4.: Metamodel-based Model-to-Model Transformations [18]

Figure 5.4 shows the high-level structure of a Model-to-Model transformation. In general, every Model-to-Model transformation is defined as a set of transformation rules. These transformation rules describe how to map from any model of a defined source Metamodel to a model of a defined target Metamodel. Figure 5.4 pictures a simplified transformation where only one source Metamodel and one target Metamodel are used. Model-to-Model transformations generally can also describe mappings from multiple sources to multiple target Metamodels. The transformation rules are defined in an executable language. A runtime environment for that language can use these rules and models of the correct source Metamodels to apply the transformation and generate target models [22, 97].

Model-to-Model transformations can be categorized as either horizontal or vertical transfor-

mations. The Model-to-Model transformations Template-of-Design and Template-of-View are both categorized as vertical Model-to-Model transformations: they use a set of source models that has a higher level of abstraction as the target models [40].

It is further possible to categorize Model-to-Model transformations as endogenous or exogenous. An endogenous transformation is a transformation where the source Metamodel matches the target Metamodel. An exogenous transformation is a transformation where the source Metamodel differs from the target Metamodel. Endogenous transformations are typically used to modify smaller subsets of an overall model [40]. The transformations Template-of-Design and Template-of-View are vertical and exogenous transformations. These transformations are however utilizing some endogenous aspects where the orthogonal object properties (see Section 4.5.1) and operators (see Section 4.5.2) are used. Moreover, some transformations are applied after the execution of the Template-of-Design which refine and complete the information provided by the user.

5.3.1. Model-to-Model Transformations based on Domain-Specific Languages

This section describes different approaches to Model-to-Model transformations used in software engineering. The goal of this section is to understand the drawbacks and advantages of these methods. This assessment ultimately helps understand why the approaches are not well suited for the design goals of the MDA flow introduced in this thesis and helps demonstrate how the Template-of-Design approach to Model-to-Model transformations does not suffer from the shortcomings.

There are many approaches to Model-to-Model transformations which are based on Domain-Specific Languages developed in particular for this purpose. The most important representatives of this are Kermeta [43] (which is based on the Xtend programming language), the ATLAS Transformation Language (ATL) [38], the XML-based eXtensible Stylesheet Language Transformations (XSLT) [50], and the Epsilon Transformation Language (ETL) [80, 45]. XSLT is noteworthy as it follows a slightly different approach to the other transformation languages: instead of defining its own custom Domain-Specific Language, XSLT relies on XML, its syntax, and XML schema. On top of these foundations, XSLT uses a template-based syntax and other XML technologies such as XPath to provide a powerful transformation toolset. Aside from this difference, the assessment conducted on these languages as part of this thesis shows that all DSL-based Model-to-Model transformation languages are very similar in their nature. As part of this work, the next section introduces the most popular Model-to-Model transformation language ETL as an example.

5.3.1.1. Epsilon and the Epsilon Transformation Language (ETL)

The Epsilon Transformation Language (ETL) is a Model-to-Model transformation language that was developed as part of the Eclipse Modeling Framework (EMF) ecosystem. It comes as

one of a set of languages in the Epsilon family that were designed for automating model-based software engineering tasks. The language is based on a common expression and statement language, the Epsilon Object Language (EOL), making the different languages such as Eclipse Generation Languages (EGL), Eclipse Validation Language (EVL), and the Eclipse Transformation Language (ETL) very similar. Being such an integrated component of the Eclipse ecosystem, it also comes with some IDE support. This is at least helpful for development teams that have not yet moved away from Eclipse-based IDEs.

```
1  (@abstract)?
2  (@lazy)?
3  (@primary)?
4  rule <name>
5      transform <sourceParameterName>:<sourceParameterType>
6      to <targetParameterName>:<targetParameterType>
7          (,<targetParameterName>:<targetParameterType>)*
8      (extends <ruleName> (, <ruleName>*)? {
9
10     (guard (:expression)|({statementBlock}))?
11
12     statement+
13 }
```

Listing 5.1: Syntax of an Eclipse Transformation Language (ETL) Transformation Rule [80, 45]

The Eclipse transformation language allows a developer to define ETL modules that contain a set of named transformation rules. Listing 5.1 shows a snippet from the grammar of the ETL language that shows how a rule is defined:

An ETL rule can be labeled as `abstract`, `primary`, or `lazy`. After this labeling rule, the rule body is defined: A rule is named and describes a source and one or multiple target model elements, defined by the keywords `transform` and `to`. To refer to one or more (potentially abstract) rules as a foundation for a newly defined rule, they can be listed after the `extends` keyword. Inside the curly braces, the rule's main body is defined. Here, an optional `guard` can be used to limit whether a rule is applicable to a certain model element or not. The set of one or more `statement`s in EOL language is then defining the actual transformation.

When an ETL module is executed, all rules which are not marked as `lazy` or `abstract` are executed. A rule is applicable to a certain model element if the `sourceParameterType` matches the type of the model element and if the guard does not prevent the rule execution. During the application, elements of `targetParameterType` are instantiated and then populated via the `statements` defined in the rule body. During this process, further rules can be recursively invoked. For this purpose, ETL provides the `equivalent` or `equivalents` operation. This operation identifies rules that can be used to construct a target model element from a source model element provided as an argument. When a transformation is requested via the equivalents operation, the execution engine also utilizes rules that are labeled as `lazy` [80, 45].

Example ETL usage An example of how ETL can be applied to real-world models is provided by the ETL documentation [80, 101] and used in the following to illustrate the concept.

Figure 5.5 shows a sample Metamodel for project planning. In this Metamodel, a Project contains a set of Tasks and Persons. Each Task of a Project has a start and duration. The effort of these tasks is then assigned to the `Persons` member of the project. For this purpose, the Effort class is defined which allows assigning a certain percentage of a Task to one project member.

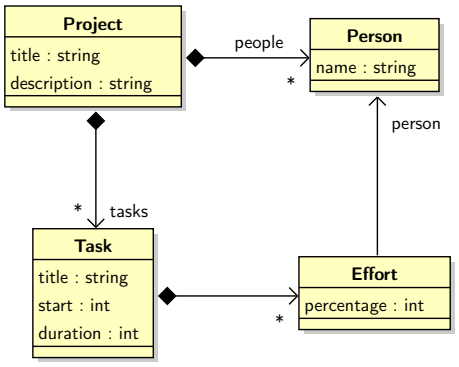

Figure 5.5.: Example Source Metamodel for ETL transformation [101]

Figure 5.6 contains a model describing one of the projects as defined in the Metamodel in Figure 5.5. In this project, the three tasks Analysis, Implementation and Design need to be performed. As per the model description, 60% of the Implementation task and all of the Analysis task is done by Alice. Bob does 40% of the Implementation task and 70% of the Design task. Charlie performs 30% of the design task.

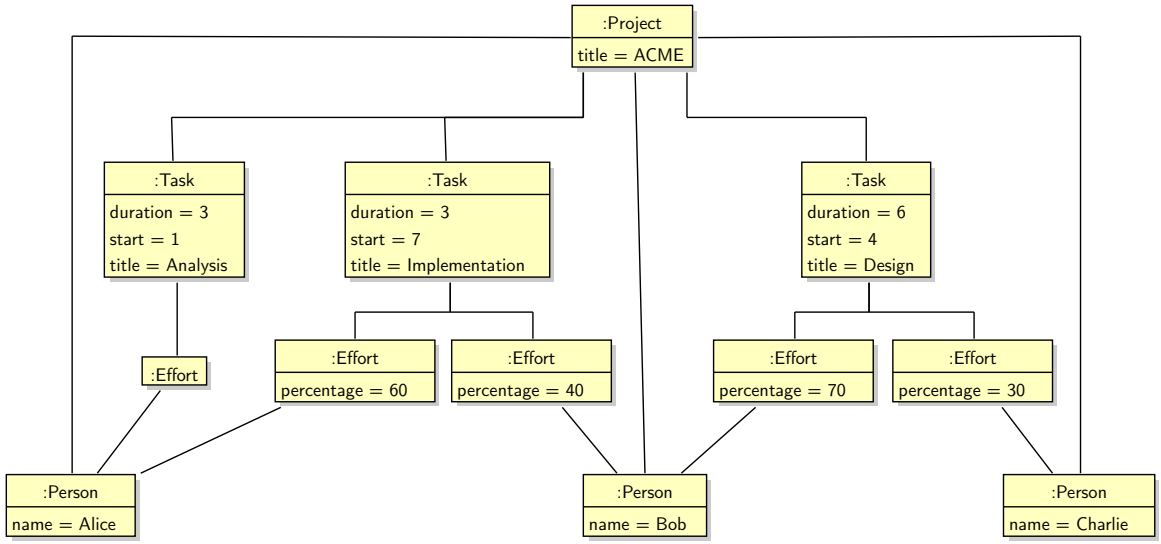

Figure 5.6.: Example Source Model for ETL transformation [101]

The Target Metamodel in Figure 5.7 is a different kind of description of the Project. Here, a

Project contains a set of Deliverables, where each deliverable has a Name, a due date, and a lead. The lead in turn is a `Person` object. This Metamodel only describes persons who hold the lead role for a Project. In this scope, a Lead is defined as the Person doing the majority of the work for a Deliverable.

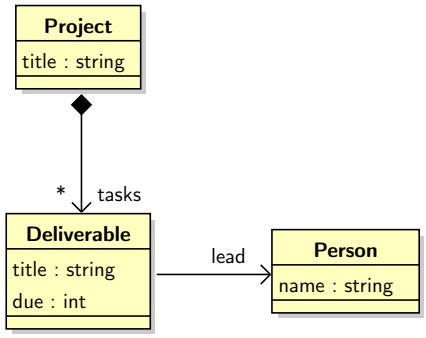

Figure 5.7.: Example Target Metamodel for ETL transformation [101]

Listing 5.2 contains the ETL transformation that will use a source model of the Metamodel in Figure 5.5 and transform it into a target model of the target Metamodel. Based on the taxonomy rules described in the introduction of this section, this transformation can be defined as an exogenous vertical transformation.

In total, the transformation in the listing contains three rules:

- The first rule `ProjectToProject` describes how a Project instance of the source Metamodel `s` can be transformed into a Project instance of the target Metamodel. The `=` operator defines that the `title` of the target project `t` will be set to the title of the source project `s`. Furthermore, the `::=` operator describes that the `deliverables` of the target project `t` have to be set to the *transformed* `tasks` of the source project `s`.

- The `TaskToDeliverable` rule describes how each `Task` of the source Metamodel is to be transformed to a `Deliverable` of the Target Metamodel. Here, some transformations are applied to the Task attributes `title` and `start` and `duration`. This is needed because the target Metamodel models a different aspect of the Project: it cares about the deliverables instead of the Tasks to get them. Deliverables are described by a `due` date, not `start` date and a `duration`. In this example, the main deliverable is always a report of the task being performed, hence the change in names. Eventually, this example applies a more complex transformation to identify the `lead` responsible for a deliverable. For this purpose, the transformation rule in line 16 uses an EOL expression to find the assignee who performs the highest percentage of the work for a task in the source model. The syntax of this expression is similar to the C# LINQ style [48] or the stream API enabled in Java 8 [65] or C++14 [64] and will not be described here in detail. The identified source task which has the highest effort however also identifies the person responsible for most of the deliverables. This `Person` of the source Metamodel is then transformed to a Person of the target Metamodel as triggered by the `::=` assignment.

```
rule ProjectToProject
        transform s : Source!Project
        to t : Target!Project
{
        t.title = s.title;
        t.deliverables ::= s.tasks;
}

rule TaskToDeliverable
        transform t : Source!Task
        to d : Target!Deliverable
{
        d.title = t.title + " Report";
        d.due = t.start + t.duration;

        d.lead ::= t.effort.sortBy(e|-e.percentage).first()?.person;
}

@lazy
rule PersonToPerson
        transform s : Source!Person
        to t : Target!Person
{
        t.name = s.name;
}
```

Listing 5.2: Example Transformation in ETL [80]

- How this is done is described by the last transformation rule `PersonToPerson`. This Transformation rule is labeled as `lazy` to ensure only Person objects are created in the target model when they are explicitly requested by a `::=` assignment in a different rule. In the given example, `Charlie` will not appear in the target model as he is not the lead of any `Deliverable`.

After applying these rules to the input model in Figure 5.6, the model pictured in Figure 5.8 is constructed. It can be seen that the attribute has been transferred for the `Project`, the `Deliverable` instances have been created for each `Task` and simple calculations have been performed for the `Deliverable` attributes `title` and `due`. Moreover, the two required `Person` instances have been created and assigned to the `Deliverables`.

This simple transformation described here can be performed in all other common transformation languages with a similar look and feel. For the sake of brevity, examples of these transformations are omitted from this thesis.

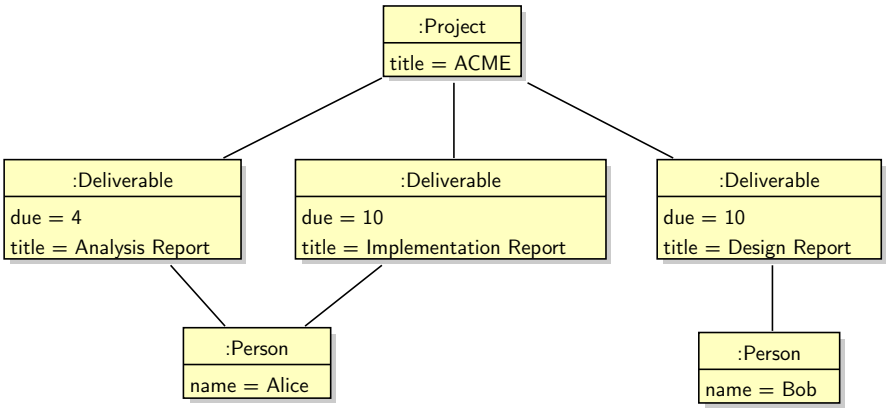

Figure 5.8.: Example Target Model for ETL transformation [101]

5.3.1.2. Evaluation Results

Model-to-Model transformation Languages available as Domain-Specific Languages exhibit the following key advantages:

1. Modularity: The specification of which elements a transformation should be applied to and what effect it should have is central to Model-to-Model transformation Languages. This inherently leads to a high degree of modularity where larger, more complex transformations must be broken down into smaller, reusable pieces with a structure similar to that of the source and destination Metamodels. This can make the transformation process easier to understand and generally simplifies re-usability: It basically enforces clean software architecture of the transformations.

2. Low Verbosity: The languages are tailored specifically to the domain of Model-to-Model transformations and it is therefore possible to express simple transformations with very little code.

3. Declarative Nature: Many of these languages support and encourage a declarative paradigm. This means it is possible to specify which kind of transformation result to be achieved, rather than specifying how the transformation is to be achieved. The runtime engine which executes the transformation language will then handle the task of transforming the what into a how: It will either try to identify a user or library-defined rule to perform the transformation or use its built-in heuristics. At the same time, the declarative description provides a view of the transformation that is at a higher level of abstraction, which can make the transformation description more readable.

The increased readability and the low verbosity of the languages show especially for vertical transformations and for transformations where the level of abstraction that is bridged is minimal.

Our evaluation however also shows some grave disadvantages:

1. Lack of Tool Support: Similar to most DSLs, Model-to-Model transformation languages are applicable only to narrow subdomains of software engineering. The user base of these languages is therefore very limited and there is little to no commercial or open-source tool support. This becomes especially problematic for more complex transformations: debugging, profiling, and linting tools are required as elements of a general-purpose software development environment. An environment for the development of generators is no different in that context and a high level of productivity cannot be reached with DSL-based environments that do not provide these tools.

2. Steep Learning Curve: All new languages come with a certain learning curve and Model-to-Model transformation languages have an especially steep learning curve. Their limited user base leads to very few available resources and their often declarative nature makes it difficult for new developers to get started.

3. Limited Expressiveness: The languages are designed with a specific set of features relevant to the domain of Model-to-Model transformations. They are mainly targeted at translating between different models of highly similar structure. To apply Model Driven Architecture to digital hardware design and other IC design domains, complex transformation rules are required. The capabilities of Domain-Specific Languages fall far short of those of general-purpose programming languages – especially when it comes to utilizing libraries.

These limitations are critical because they prohibit achieving several of the overall goals of this thesis: First, these languages are not suited for complex transformation rules that need to bridge a high level of abstraction – which is clearly required to bring code generation to new fields of IC design. Second, the framework developed as part of this thesis needs to be used by engineers who do not have a strong background in software development – a steep learning curve is irreconcilable with this goal. Third, an efficient development of generators using MDA means efficient development of transformations – a high level of community support and available resources are considered as a prerequisite to develop and debug transformations efficiently. It is therefore clear that the approach toward Model-to-Model transformations taken in this thesis must not rely on any Domain Specific Languages.

5.3.2. Suggested Approach

Instead of relying on custom Domain-Specific Languages for Model-to-Model transformations, this thesis embeds the concept of Model-to-Model transformation into the Python programming language. Python acts for Model-to-Model transformation as an Embedded Domain-Specific Language (EDSL) as introduced in Section 3.3.

Many of the advantages identified for DSL-based Model-to-Model transformations are not limited to true Domain-Specific Languages as the same behavior can be achieved with Model-

to-Model transformation libraries using Python as an Embedded DSL. Libraries such as the popular AutoMapper [99] library demonstrate how Model-to-Model transformations defined in an Embedded DSL can provide high readability and low verbosity. It provides transformations with very little code and high readability especially if the source and target models are similar. In an Embedded DSL context, libraries like AutoMapper can be selectively applied only where the models which shall be mapped match these criteria and make the approach suitable. Moreover, the use of object-oriented programming and state-of-the-art software architectural patterns can provide a high degree of modularity of the Python-based Model-to-Model transformations.

The disadvantages of Domain Specific Model-to-Model transformation Languages listed above do not exist when using Python as a language for Model-to-Model Transformations: developers work in a language ecosystem with excellent tool support, benefit from a particularly flat learning curve, and can benefit from the full expressiveness of the Python language and the vast ecosystem around it. In short, the approach benefits from all the advantages the Python language already brings for the entire Infineon Metamodeling Framework also in the area of Model-to-Model transformations (see Section 5.1.1).

These Model-to-Model transformations are referred to as *Templates* because they act as blueprints or generators for instances of the underlying Metamodels. The Template-of-Design (ToD) for example is the blueprint that generates the Model-of-Design (MoD) and the Template-of-View (ToV) is the blueprint that generates the Model-of-View (MoV). In this context, it is important to emphasize that the template itself is not a single file that describes the transformation. It can be an arbitrarily complex software module, with its own submodules which can be combined, configured, and modified in a flexible manner. Essentially, the ToD is pure Python code that can use all the languages' features and the APIs automatically generated for the Metamodel. It is especially not to confused with the templates processed by template engines such as Python's Jinja or Mako for target code generation.

Figure 5.9 shows how the Model-to-Model transformation process works. The transformation itself is a Python script that runs on the same Python interpreter that also holds the APIs generated by the Metamodeling framework for the source and target Metamodel. The transformation script then accesses the APIs of one or multiple source models (which may be of one or multiple different Metamodels) and *pulls* the model information from it. It then processes and transforms this information and eventually *pushes* the information to the target model through the API generated for the target Metamodel.

5.3.3. Efficient Template-of-Design Libraries and APIs

The APIs that Infineon's Metamodeling Framework generates for the provided Metamodels were initially designed to be used mainly in Mako and Jinja templates or in custom readers and writers. In particular, these APIs restrict access to the constructor of the classes defined in the Metamodels. Consequently, all model artifacts always need to be constructed as a tree in a "top-down" manner from its root node. This makes the APIs ideally suited to develop

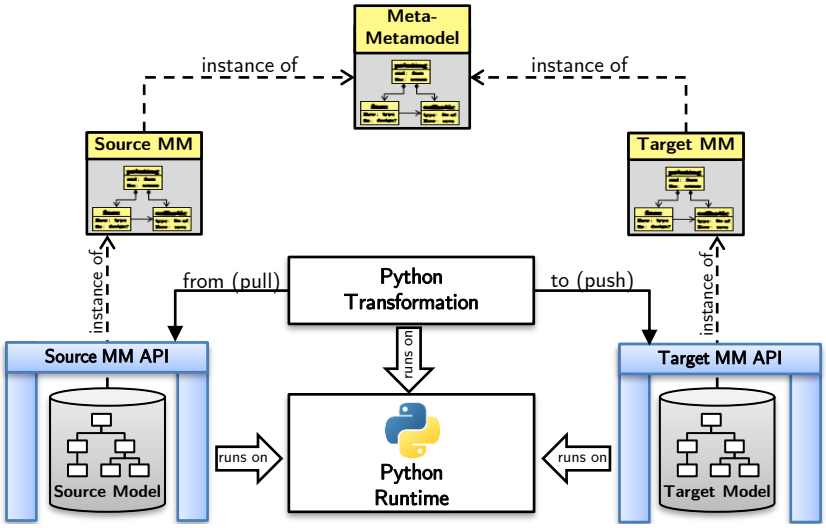

Figure 5.9.: Python Model-to-Model transformation based on Metamodels and Metamodel APIs

readers and writers and for utilization in template engines in simple Metamodeling-based code generation as described in Section 2.3.2.

Using these APIs for the implementation of a Template-of-Design, however, comes with some limitations. Section 4.3.3 has already demonstrated based on the FIR filter example that even for simple MoT instances, the MoD representation of a model can become quite complex. To demonstrate the implications of this, an excerpt from the Model-of-Design described in this section, which is shown in Figure 5.10 is used. It contains the elements from the top left of the MetaRTL MoD of the FIR filter instance described in Section 4.3.3: the input port `data_in` of the `FirFilterInstance` module, first Register the input data is stored in and the connection between the input port and is registered. It is important to note that this part of the MoD is static, i.e. not dependent on the configuration provided by the MoT instance as any input data needs to be registered first to avoid timing issues when the module is integrated into the system context.

Listing 5.3 shows the ToD code that creates this static instance using the default APIs generated by the Infineon Metamodeling Framework. The advantage of this approach is that it is clearly visible to the developer that he is working on the underlying model. It is however quite clear that this style is not practical, especially for a static part of the design that does not depend on parameterization. Moreover, both examples do not contain object properties that need to be added to each of the ports for real-world use. Adding these elements further expands the model and code size and thus reduces the practicality of this approach.

Based on this example, it is possible to motivate and describe several changes and extensions made to the APIs generated by the Infineon Metamodeling Framework. Moreover, library elements have been developed to make the creation of models more efficient. The main goal of these extensions is to make the Template-of-Design as efficient or almost as efficient as that

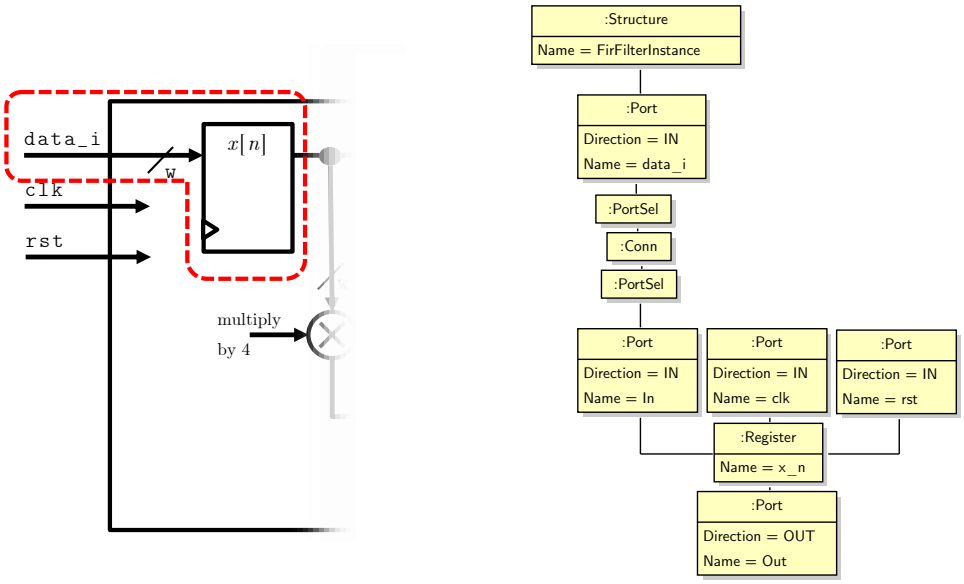

Figure 5.10.: Excerpt from MoD used in Section 4.3.3

of state-of-the-art Hardware Generation Languages introduced in Chapter 3, however without hiding the fact that the applied approach is clean and model-based. It is important that at any time, the developer of the Template-of-Design should be aware of the fact that he is actually only populating a model.

5.3.3.1. Access to Object Constructors

The first limitation that has been removed as part of this work was the restricted access to object creation via class constructors. In the Infineon Metamodeling Framework, any object has to be created via `parentObj.createX()` or `parentObj.addX()` method calls.

This is problematic as it prevents the usage of several best practice techniques in software engineering. The first important technique is the application of the Liskov substitution principle (the L in SOLID), also referred to as strong behavioral subtyping. This principle states that object-oriented software has to be designed in a way that instead of passing an object of a type `BaseType`, it always has to be possible to pass an object of a type `DerivedType` as long as `DerivedType` inherits from `BaseType`. This principle is closely tied to the Open-closed principle (the O in SOLID), which states that "[s]oftware entities [...] should be open for extension, but closed for modification" [12]: any derived type must still adhere to the contract defined for the base type. Derived types may add or extend the base type, but must never modify the behavior of the base type.

In the context of the Metamodel-generated data structure, the code has been modified so that objects of classes defined by Metamodels can be instantiated on their own, outside of the auto-generated `createX()` or `addX()` methods. These objects are then automatically added to the

```
def createFirFilter(parentStructure, firFilterMoT):
    firFilter = parentStructure.addComponent("Structure")
    firFilter.setName(firFilterMoT.getName())

    data_i = firFilter.addPort()
    data_i.setDirection("IN")
    data_i.setName("data_i")

    data_i_reg = firFilter.addComponent("Register")
    data_i_reg.setName("data_i_reg")
    clkPort = data_i_reg.addPort()
    clkPort.setDirection("IN")
    clkPort.setName("clk")
    rstPort = data_i_reg.addPort()
    rstPort.setDirection("IN")
    rstPort.setName("rst")
    inPort = data_i_reg.addPort()
    inPort.setDirection("IN")
    inPort.setName("In")
    outPort = data_i_reg.addPort()
    outPort.setDirection("OUT")
    outPort.setName("Out")
    data_i_reg.setSyncRstRef(rstPort)
    data_i_reg.setClkRef(clkPort)
    data_i_reg.setInRef(inPort)
    data_i_reg.setOutRef(outPort)

    conn = firFilter.addConn()
    conn.addPortSel().addPortRef(data_i)
    conn.addPortSel().addPortRef(data_i_reg.getInRef())
```

Listing 5.3: Creation of static MoD excerpt shown in Figure 5.10 using default API

respective parent object by their constructor or can be manually inserted using a set of newly added `parentObj.insX(createdObj)` methods.

With this new feature, the Liskov substitution principle can be applied in the Template-of-Design based on the Infineon Metamodeling Framework: instead of creating a method `createFirFilter`, a new class `FirFilter` can be created. This new class is derived from the MetaRTL `Structure` and contains equivalent code in its constructor.

Moreover, this change significantly improves testability. In general, mixing behavior such as object creation, object deletion, data validation, and other checking with data structures is considered bad practice. Wherever this is necessary, it is important to separate software units

(for example classes) from one another as much as possible. The Weak Coupling Principle states that when software units communicate, they should exchange as little information as possible. As shared state is just a form of communication, this also implies that separate objects should have as little shared state as possible. Applying this principle makes it possible to create and test objects on their own, independent of e.g. the parent object that they are linked to and independent of the context in which they exist. This change is therefore also a fundamental modification to enable clean unit testing [77].

5.3.3.2. Addition of Default Constructor Behavior

The previous section describes how inheritance can be used to create derived classes of the MetaRTL `Structure` which are then populating the internal of the structure with registers and combinational logic, implementing generators for certain defined modules such as the `FirFilter`. This concept is not only used for custom `Structure` subclasses. Instead, it is used to add default child objects to many common MetaRTL objects on their creation. Lines 11-26 of Listing 5.3 contain the default ports that are required for every Register. The MDA framework implements default constructors for all components defined by MetaRTL. These constructors create the object attributes that are required for a component and set the correct associations. When the constructors exposed to the user of the Template-of-Design are used, it is therefore not possible to forget manually adding the ports or to forget calling a library function that does it – the required minimum is automatically created.

5.3.3.3. Flexible Support for Constructor Parameters

Describing generators in the Template-of-Design format is an extremely powerful and flexible way to describe configurable designs. The configurability and flexibility of this approach is however usually only needed in a subsection of the design. This is well illustrated by 5.3, which describes a subset of the FIR filter which is always the same, regardless of the MoT instance that is passed. When describing static elements, a concise notation is particularly important. The extensions added to the default constructors in the MetaRTL API provide this notation in different ways.

```
1 notationA = Structure(Name='FIRFilter',
2                       Ports=[Port(Name='data_i', Direction='IN'),
3                              Port(Name='result', Direction='OUT')])
4 notationB = Structure(Name='FIRFilter',
5                       Ports=[{'Name': 'data_i', 'Direction': 'IN'},
6                              {'Name': 'result', 'Direction': 'OUT'}])
```

Listing 5.4: Supported Notations for Object Creation

Listing 5.4 shows these notations. First of all, it is possible to pass attribute values directly into

the constructor using named arguments as used in both notations (`Name='data_i'`). Moreover, the new notation also allows us to directly pass in composition objects (see `notationA`) or to create them and set their attributes using their default constructors (see `notationB`).

5.3.3.4. Library Routines for Efficient Connectivity

On a high level, the Template-of-Design defines `Structures` and their interfaces, instantiates sub-structures, inserts combinational logic, and crates connectivity. A significant part of the ToD code is responsible for creating connections. It is important for the MetaRTL Metamodel to support connections as actual first-class artifacts with connections being their own classes as this facilitates metaprogramming on connections (it for example makes it possible to pass connections to methods). Despite this, the common use-case of connecting one port to another needs a simple solution without having to deal with the `Conn` objects that need to be created as defined by the Metamodel.

For this purpose, a connectivity library was implemented that adds `source.connect(target)` to every source object of type `Port`, `Conn` and `Literal` and supports target objects of the same types. When this method is used, the corresponding Metamodel structures are transparently generated.

5.3.3.5. Automated Clock and Reset Connectivity

The ToD can be kept simple since the framework has a powerful resolution mechanism to create the final connectivity and infer attributes needed to generate views from the MoD. One important aspect of this resolution mechanism is its ability to connect the clock and reset ports for stateful elements.

Using a Model-to-Model transformation on the Model-of-Design, these ports are automatically connected to the clock and reset ports with are closest in the hierarchy and have attributes that label them as compatible clock and reset ports. This is particularly helpful for design hierarchies that have only one clock and reset domain. A key property of this Model-to-Model transformation is that it will not override any existing connection. It is therefore easily possible to manually provide the clock and reset for a few modules which differ in their clock and reset domain and let the Model-to-Model transformation handle the majority of the connectivity. Of course, all connections and ports can be made explicit as the transformation can be either skipped or, even if applied, does not modify any existing connectivity.

5.3.3.6. Object Property Propagation

Another resolution mechanism is provided to identify object properties for ports and connections for which they are not defined in the Template-of-Design. For this purpose, a Model-to-Model

transformation is provided that modifies the Model-of-Design. This transformation derives object properties for all ports and connections based on information already partially available in the Model-of-Design.

The mechanism is a relatively simple forward-first propagation mechanism. To support this mechanism, all blackbox components (components other than the MetaRTL `structure`) provide their own library methods that are used to calculate object properties for one of their ports depending on the other ports. The algorithm will first use the available object properties on output ports and propagate them along the available connections, considering slicing and other connectivity features. For every newly identified port property, the resolution algorithm either forwards it along the connectivity inside the structure or calls the library methods of blackboxes to check whether this available input property has an impact on output object properties. This is repeated as long as new information either about the component's ports or connections can be found. As soon as the forward propagation (along the signal direction) is completed and no more information is found, a backward propagation is triggered, where every object property identified using the backward propagation invokes the forward propagation again.

Like this, an approach is available that meets some criteria that are important for intuitive understanding: all manually set object properties are never overwritten – if a developer decides to set incorrect properties by hand, it will be caught in the Model-of-Design data validation. All forward propagation has a higher priority than backward propagation. If any incompatibility is identified by backward propagation, the information obtained from it is not used and an error appears during data validation, forcing the developer to manually correct the mistakes in his ToD. In the end, either all components are fully configured – including the availability of all ports – and all wire sizes are known and compatible, or an error message is shown summarizing the items which were either conflicting or missing during the data validation on the model.

As part of this thesis, the use of a formal SMT solver [44] was investigated to resolve some more items and provide object properties that would meet potentially conflicting requirements. This analysis however showed that it is not worth the effort, since the number of additional resolutions is small, complexity issues pop up and the reports from solvers are hard to understand. In general, the approach where incompatibilities are highlighted during the checking of the model is easy to understand and work with. Even without the use of these advanced methods, the identification of object properties rarely requires manual support on the ToD side, the rationale for the properties that need to be provided is typically clear for the developer as the errors highlight information that is actually missing from the model.

5.3.3.7. Results

The changes and extensions described above, as well as a few additional modifications such as the replacement of getters and setters such as `obj.getValue()` and `obj.setValue(42)`) with properties `obj.Value` and `obj.Value = 42`) that transparently access the corresponding setter and getter underlying the call has led to further improvements in readability.

Listing 5.5 shows how the equivalent of the original snippet in Listing 5.3 looks when making use of the improvements described above and moving it into the constructor of a dedicated child class of `Structure`. Aside from the improved readability the object-oriented approach also opens up the possibility for a clean and extendable architecture of all ToD code, guaranteeing a high degree of reuse.

```
1 class FIRFilter(Structure):
2   def __init__(self, firFilterMoT):
3     super().__init__(Name=firFilterMoT.Name,
4                      Ports=[{'Name': 'data_i', 'Direction': 'IN'}])
5
6     data_i_reg = Register('data_i_reg')
7     self.Ports['data_i'].connect(data_i_reg.InRef)
```

Listing 5.5: Creation of static MoD excerpt shown in Figure 5.10 using extended API

In terms of visual impression, the ToD code using object constructors and property-style attribute access instead of getters and setters has a lot of similarity to the style common for structural descriptions of designs in state-of-the-art HDLs such as SystemVerilog and VHDL. This similarity has made the adaptation of the MDA methodology easy, particularly in an environment of hardware designers familiar with Python scripting.

It is also important to note that the API modifications were performed using the metaprogramming capabilities of the Infineon Metamodeling Framework. These improvements are therefore not just available for MoT, MoD, and MoV as part of the MDA flow presented here but can easily be ported over to other Metamodels.

5.3.4. Transformation of Model-of-Things Input Models

To illustrate the transformation methodology that was developed without the complexity of real-world applications, this section shows a Template-of-Design which derives a digital filter from the Model-of-Things sketched in Section 4.2. The Template-of-Design is built for the Metamodel in Figure 4.2 and will construct MoD instances for all models of this Metamodel. Figure 4.10 contains a block diagram of the circuit it creates for the sample model in 4.3.

In addition to describing circuits in the static HDL netlist style, the complete feature set of Python can be utilized. This provides more flexibility to define the architecture. Section 4.2 already named the utilization of Python-based scientific computing libraries to derive parameters for certain microarchitectures. In extreme cases, the Template-of-Design can even run instances of the complete MDA toolchain to find or evaluate solutions. Here, code is generated and analyzed in order to find out the right – or also the best – way to generate design items.

The Template-of-Design source code is pictured in Listing 5.6. When executed, line 34 adds a `FIRFilter` instance to the Model-of-Design. The construction of this `FIRFilter` takes place

```
1  class FIRFilter(Structure):
2    def __init__(self, firMoT, parent):
3      super(). \
4        __init__(Ports=[{'Name': 'data_i', 'Direction': 'IN'},
5                        {'Name': 'result', 'Direction': 'OUT'}],
6                 parent=parent)
7      current_delay, sum_conn = 0, None
8
9      current_conn = Connection(self['data_i'])
10     for addend in sorted(firMoT.Addends,
11                          key=lambda x: x.Instant):
12       while current_delay < addend.Instant:
13         reg = Register()
14         current_conn.connect(reg.In)
15
16         current_conn = Connection(reg.Out)
17         current_delay += 1
18
19       mul = HWMUL(Constant=addend.ImpulseResponseReal)
20       mul.addIn().connect(current_conn)
21
22       instant_out = Connection(mul.Out)
23       if sum_conn is None:
24         sum_conn = instant_out
25       else:
26         adder = HWPLUS()
27         adder.addIn().connect(instant_out)
28         adder.addIn().connect(sum_conn)
29         sum_conn = Connection(adder.Out)
30
31     sum_conn.connect(self['result'])
32
33 global fir_MoT_1 # instance of a FIRFilter MoT is provided by Framework
34 myFilter = FIRFilter(fir_MoT_1, toplevel)
```

Listing 5.6: Template-of-Design example for the generation of n-th order FIR filter from Model-of-Things

in the constructor in Lines 2-31. First, the Ports of the filter are defined. The example here only defines the ports `data_in` and `result` and omits the clock and reset lines necessary for the microarchitecture. These are later inserted by a transformation on the Model-of-Design or by applying automatic connectivity resolution. In Line 9, every execution of the ToD instantiates a connection in the Model-of-Design. This connection is attached to the `data_in` port of the filter. Lines 10-29 contain the part of the filter that depends on the Model-of-Things. Here,

the ToD iterates over all `Addend` instances of the Model-of-Things, sorted by their `Instant` attribute in ascending order. Lines 12-17 contain a while loop including the loop body which is executed to insert delay registers. The sample MoT uses values of consecutive instant n, $n-1$, $n-2$ for every output sample $y[n]$, the loop is thus executed once per iteration of the enclosing for-loop. A multiplier is then inserted in line 19 and the connection referenced by `current_conn` is attached to the multiplier in line 20. In the first iteration of the for loop (line 10-29), `sum_conn` is then set to reference the connection object which is attached to the output of this multiplier. In every further iteration, an adder is inserted which sums up the connection previously referenced by `sum_conn` and the output of the multiplier. After this, `sum_conn` is redirected to point to a connection attached to the output of the adder. After all loop iterations are completed, line 31 attaches the `result` port of the FIRFilter component to the connection referenced by `sum_conn`.

It is important that the Template-of-Design is only one possible micro-architectural template. For the same Model-of-Things, it would be feasible to develop a ToD that generates an adder-tree-based microarchitecture or a multicycle filter using a multiply-accumulate unit. The flexibility of the Template-of-Design approach based on Python becomes clear when performing a closer assessment of this example: the tradeoff between different micro-architectural templates depends on the concrete Model-of-Things and its parameter properties. While the MoT example with the multiplicative factors of 2 and 4 are ideally suited for single-cycle multiplication, other MoTs might require a different microarchitecture to provide meaningful synthesizable hardware. This analysis can be done automatically in the ToD itself. A library for the analysis of the MoD can determine which of the available micro-architectures best fits the provided parameter set as well as potential timing, power and area constraints that can be provided by a separate Model-of-Things and then automatically instantiate the correct microarchitecture.

5.3.5. The Language and Modeling Nature of the Template-of-Design

It is subject to debate whether the Template-of-Design is itself a language based on MetaRTL or whether it is simply a Model-to-Model transformation with significant convenience added on top.

The Template-of-Design approach and the MDA methodology presented here is in the meantime used for many designs at Infineon Technologies AG. These designs are in detail described in Chapter 6.3 and have a significant size, making up entire subsystems of complex ICs. An analysis shows that the readability and efficiency of input in the Template-of-Design are comparable to the state-of-the-art hardware generation language approaches.

Manually developed library components that are provided in addition to the MetaRTL API and which are described in detail in Section 5.3.3 cannot all be classified as Model-to-Model transformation on the MetaRTL models. For example, the `connect` functions clearly meet the criteria for a library and not for a Model-to-Model transformation.

Nevertheless, the modifications that were done on the default APIs that are provided by the

Infineon Metamodeling Framework are relatively minor and do not hide a significant part of the underlying Metamodel. This especially applies to design engineers who are familiar with the Metamodeling concept, who are aware of the Python-based code generation it is used for, and who have used the Infineon Metamodeling Framework to generate single source views. These engineers are immediately able to identify the commonalities between the Template-of-Design and the APIs provided for view generation. With this background, they still perceive their development of the Template-of-Design as the development of a model transformation. Moreover, the easiest learning path to understanding the MDA flow presented in this thesis and the development of Templates-of-Design is still based on the understanding of the Metamodels involved.

In summary, the presented method is mainly a Model-to-Model transformation approach that is enhanced by library elements that abstract the underlying Metamodel in the style of an Embedded DSL only where necessary.

5.4. View Language Description and Automated View Generation

After all Templates-of-Design and transformations on the Model-of-Design are executed, the design-centric Model-of-Design is available. This Model-of-Design can be further used for analysis and transformations. Eventually, it will be used to generate target views. In a traditional Metamodel-based flow as described in Section 2.3.3, this path usually relies on code generators based on template engines. The development of these templates is challenging and gets increasingly difficult the more abstract the input specification becomes. This phenomenon has been described as the *Generator Gap* (see Section 2.3.6. A significant part of the Generator Gap is also caused by the challenges of generating syntactically correct, human-readable, and configurable target views. The MDA-based approach addresses these challenges on the Model-of-View layer.

The following describes the tasks that need to be taken care of here and the problems these tasks cause when they are solved with template engine-based code generation:

Syntactical correctness of the target view needs to be ensured. This can be difficult for the developer as he cannot rely on the syntax highlighting of editors as the generated views are partially made up of control statements of the template engine and partially of the grammar of the target view. Moreover, when handling lists of elements, the first or last element of the list often needs to be treated differently from all other elements. Good examples of this are the port lists of VHDL entities and SystemVerilog modules: while all ports need to be separated by a comma, the last port may not be followed by a comma. The same applies to arguments for functions and methods in many software programming languages. Another example is that empty generic and port lists are not permitted in VHDL and the corresponding `generic` or `port` keyword must be omitted in this corner case.

Formatting Human readable target views are important as the teams using the tools processing these target views need to be able to understand the content and debug the views if their usage does not yield the expected results. The templates, therefore, need to take care of proper indentation, sometimes even with indentation rules differing depending on the consumer of a view.

Configurability Different customers and different projects require different coding styles, formatting rules, and naming conventions. In the example of VHDL and SystemVerilog, the naming conventions for signalswires, instances, ports, and other artifacts can differ. Embedding these alternatives into the templates generating the views leads to complex, hard-to-read, and difficult-to-modify code.

This thesis pioneers generators based on the View Language Description (VLD) as a novel approach to view generation that resolves or significantly reduces these issues. Section 5.2 already showed how this approach is utilized as a part of the MDA flow based on the Infineon Metamodeling Framework. Figure 5.3 shows how the MoV layer's Metamodel and the actual view generation are completely automatically generated from the View Language Description format. This reduces the effort of view generation from developing a complete template to providing the Template-of-View Model-to-Model transformation.

Figure 5.11 shows Template-based and VLD-based View Generation in comparison. The VLD-based View Generation on the right shows how the view generation from Model-of-Design over Model-of-View is implemented in the MDA flow presented in this thesis. It is important to note that this approach does not depend on the MDA flow developed here. It can be utilized for all kinds of Metamodeling-based view generation and can act as a plug-and-play replacement for the Template-based view generation shown on the left part of the figure.

The benefit of VLD-based view generation is emphasized by the color coding of the figure. Here, all elements that a developer of a generator typically needs to address are colored in yellow. For both template-based generation and VLD-based generation, this involves the definition of the Metamodel and the provisioning of model data.

When it comes to the actual generation of view, the template-based view generation requires the developer to write a template that is then processed by a template engine. When doing this, all the tasks mentioned above need to be taken care of at the same time, turning the development into a complex endeavor. In VLD-based view generation, this complex task is broken down into multiple, partially reusable pieces. Moreover, a sizeable portion of these pieces is automated.

The first piece is the development of a View Language Description, an EBNF-like format describing the grammar of the target view, the intended formatting, and coding conventions. This description is used, among other things, for the automated generation of the Metamodel-of-View, a Metamodel with an AST-like structure describing the target view. Along with this autogenerated Metamodel-of-View, the APIs and infrastructure of the Metamodeling Framework can also be autogenerated.

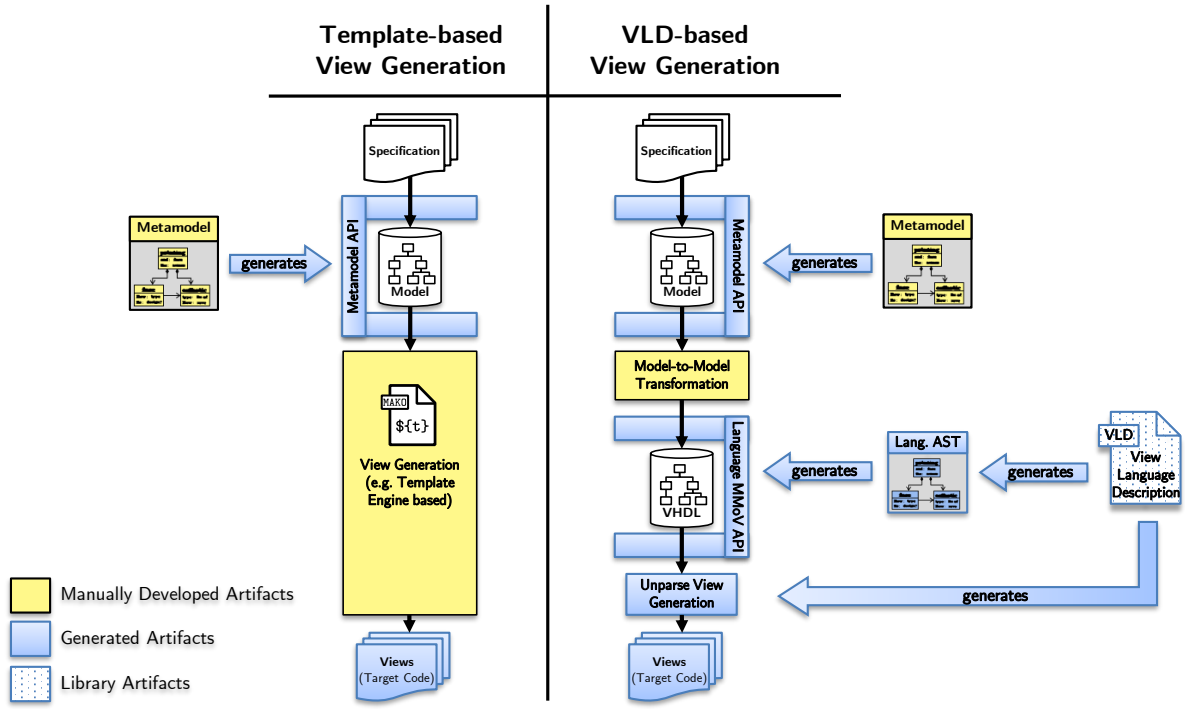

Figure 5.11.: Template-based and VLD-based View Generation Compared

Based on this Metamodel and the Metamodeling Framework tailored for it, the second piece of the VLD-based view generation is developed: the Model-to-Model transformation which takes the input model and uses it to fill the Metamodel-of-View.

The third piece of the VLD-based view generation is the generation of the actual view, referred to as Unparse View Generation mechanism. Similar to the generation of the Metamodel-of-View, this step is also completely automated and exclusively relies on the description of the target language and the intended formatting provided by the VLD format.

The following section introduces the View Language Description format and explains how it is used to both generate the Metamodel-of-View and also the Unparse View Generation code. The next section then illustrates how the Template-of-View transformation is applied and visualizes the significant effort reduction the VLD-based view generation provides.

5.4.1. The View Language Description Format

Listing 5.7 contains a simplified example of a View Language Description (VLD). The similarity between this description and an Extended Backus-Naur Form (EBNF) description is apparent. For comparison, Listing 5.8 contains a snippet from the VHDL rule for `entity_declaration` in EBNF format.

```
Entity ::= 'ENTITY ' <Name> ' IS\n'
                [Ports]
           'END ' <Name> ';\n';
Ports  ::= $indent('\t')$('PORT(\n'
                $indent('\t')$(+Port%[0:-2]: ';\n';
                                    [-1]  : '\n'  %+) ');\n');
Port   ::= $ta(1)$<Name> ' : ' $ta(1)$<Mode> ' ' $ta(1)$<Type>;
```

Listing 5.7: Simplified Snippet from View Language Description of the VHDL MoV [5]

```
entity_declaration ::=
    ENTITY identifier IS
        entity_header
        entity_declarative_part
    [ BEGIN
        entity_statement_part ]
    END [ ENTITY ] [ entity_simple_name ] ;
```

Listing 5.8: VHDL entity declaration from VHDL'93 [92]

Similar to EBNF, the description consists of a list of production rules where each of these rules in turn consists of a set of terminal or non-terminal symbols. The main goal of EBNF is to describe grammars of formal languages and thus to provide the rules for distinguishing the grammatically correct code of a certain formal language from incorrect code. As outlined in the previous section, VLD has slightly extended goals: it is the starting point to generate the Metamodel of the Model-of-View and the unparse method. This introduces three new requirements for the format. The following enumeration describes these requirements and shows how they are implemented in the VLD format.

1. While EBNF is mainly intended to describe the formal rules of a certain language, the VLD was designed to allow the automated generation of a concise Metamodel and thus an *intuitive* API. Here, it is not sufficient that all legal instances of this Metamodel form legal view instances. Instead, the Metamodel should be shaped so that it is easy to use for the developer. Artifacts that are only grammatically relevant but semantically irrelevant should not be part of the Metamodel. An example of such an artifact is the fact that in comma-separated lists, the last element is usually not followed by a comma. The Metamodel shall provide a list in this case and the generator shall treat the last element independently from the others.

2. While EBNF is structured to describe the grammar of languages and to easily and efficiently build parsers, the structure of VLD has to primarily provide a Metamodel which is used by the Model-to-Model transformation and the Unparse View Generation. The structure of VLD thus differs from the EBNF of the same language. To end up with

an intuitive Metamodel, the VLD entity directly contains `Ports` (see Listing 5.7 in Line 2) and `Generics` as well as the `LibraryClause` and `UseClause` (these are not part of the simplified VLD example provided in the Listing). The VHDL EBNF grammar that describes the entity declaration shown in Listing 5.8 is structured quite differently. Here, `library_clause` and `use_clause` are defined in the `context_clause`, which is nested a few levels inside the `entity_declarative_part` in Line 4 of Listing 5.7. The EBNF structure makes it easy to reuse code in any parser autogenerated from the EBNF or developed according to it and helps the understanding of the target language. Using the same structure for the VLD would lead to many additional classes in the Metamodel (one class is generated per rule). For the developer, this means that in the transformation building of the Model-of-View, many additional objects have to be generated, which reduces the convenience of the approach.

3. It has to describe the intended formatting of the target view. The VLD thus describes exactly one correct target view belonging to each Model-of-View instance. In contrast, EBNF grammars do not specify things like whitespaces and indentation. From the parser's point of view, any number of identical views with different formatting will map onto the same Abstract Syntax Tree. From the code generator's point of view, one formatting option has to be defined when generating the code.

To meet these requirements, several formalisms are introduced into the VLD. When the Metamodel for a VLD format is generated, a Metamodel class is added for every production rule of the VLD descriptions. For every non-terminal symbol, a rule consists of, associations are added to the sub-class. An EBNF rule for a list of at least one name would be similar to `names ::= firstName, { ', ', otherNames}`. This clearly describes that there may be either one name or a list of comma-separated names. What it does not convey is that these items, `firstName` and `otherNames` belong together semantically. When generating a Metamodel for that `names` rule, a `names` class would be inserted which would then contain two separate attributes one with the multiplicity `1` and the other one with a multiplicity of `0..*`. A developer working on the transformation from MoD to MoV would then have to take into account whether the `firstName` attribute is already set whenever he tries to add further names to the `nameList`. To avoid the accompanying overhead, a `+...+` symbol is introduced into VLD. The use of this symbol creates one attribute of multiplicity `1..*`. As typical views still frequently require different generated view code for corner cases such as the first or last element in a list, a further artifact is introduced into VLD. This artifact is used in the `Ports` rule of the VLD in Listing 5.7. Here, the `%...%` notation inside of the `+...+` block indicates that the last port (identified by `[-1]`) has to be treated differently from the others (identified by `[0:-2]`)) during view generation.

A further extension in the VLD format is the distinction between non-terminal symbols that create attributes in the Metamodel (encapsulated by `<...>`) and symbols that create compositions (not encapsulated `<...>`). Moreover, the `:` operator is used to describe both attribute name and attribute type. When a rule contains e.g. `<Size:int>`, an attribute named `Size` of type integer will be added to the class generated for the rule. If omitted, the default type is string.

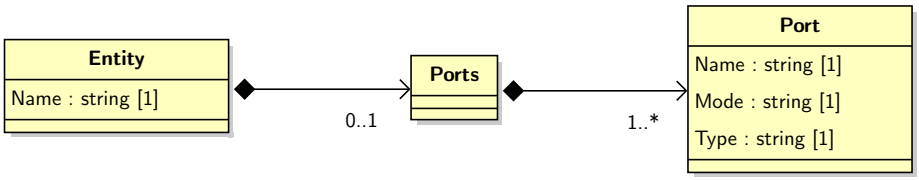

Figure 5.12.: Metamodel Generated from the VLD in Listing 5.7

Figure 5.12 shows the Metamodel that is generated from the VLD Example in Listing 5.7. It shows that for every rule, a separate class is created. For every unique attribute added to the VLD (`<...>`), a corresponding attribute is created in the Metamodel. In this case, only the `Entity` and `Port` classes have attributes. The tooling implemented further adds the compositions from `Entity` to `Port` (Multiplicity `0..1`) and from `Port` to `Ports` (Multiplicity `1..*`). Upon closer inspection, the `Ports` class of the Metamodel seems redundant. It was just included in this example for didactic reasons. In a productive use-case, it makes sense to embed the individual ports directly into the `Entity` rule and to conditionally generate the `'PORT(\n'...  ');\n'` using the `%%` notation to trigger different behavior for the first and last port.

5.4.2. Target Code Indentation and Tabular Alignment

A naïve approach to formatting target code in a nice way is inserting terminal strings containing whitespaces into the VLD. The problem with this approach is that it cannot handle indentation correctly as many production rules can occur at different levels of indentation: a concurrent signal assignment can e.g. take place at the architecture level, inside a process, or inside any number of nested conditionals, each requiring a different level of indentation. The solution to this problem is the introduction of *formatting directives*. These directives are directly introduced into the view language description. They are implemented as Python code which is included in auto-generated post-processing code by the tooling. A set of predefined directives for correct indentation of code, line breaks at certain line widths, and correct alignment of neighboring lines is provided.

The first important directive is the `indent` directive. `$indent('\t')$` in Listing 5.7, line 4 ensures an additional indent of keyword `PORT` and the following parts by one tabulator. Similarly, `$indent('\t')$` in line 5 ensures, that each port item is indented one tabulator further. This indentation applies to the element itself and all sub-elements of nested VLD rules. It is the default directive that is used to provide the indentation for blocks, conditions, classes, namespaces, processes, and similar constructs in programming languages. The effect of the indent directive in the `Ports` rule can be seen in the example in Listing 5.9 in Lines 2-5, where the first tabulator is caused by it and in Lines 3-4 where the second tabulator is caused by it.

The second important directive is the tabular alignment directive `ta`. It is used in the example in Listing 5.7 for the port list defined by the `Port` rule in Line 7. The effect of this can be

```
1 ENTITY sample IS
2         PORT(
3                 short_port_name_i␣␣␣␣ : in ␣std_logic_vector(31 downto 0);
4                 very_long_port_name_o : out std_logic
5         );
6 END sample;
```

Listing 5.9: Sample Entity showing the effect of `indent` and `ta` functionality

seen in the sample output in Listing 5.9 when comparing Lines 3 and 4: tabular alignment ensures that all the lines are properly aligned relative to one another and the port names, the port directions, and the port types are aligned as if the output was part of a table. The tabular alignment feature of the VLD tooling provides supports multiple tables per view. The insertion of the `$ta(1)$` statement tells the VLD that the following VLD elements form a new column. The rows of this table do not necessarily have to be adjacent to one another. It is also possible to insert other lines without any tabular alignment, for example containing comments in between the aligned table rows. Moreover, it is possible to define multiple tables which may even be interleaved. The argument to the tabular alignment directive defines which table the column and therefore also the currently written row is part of.

5.4.3. Template-of-View Transformations

With an existing VLD definition for a given target language, the task of view generation is reduced to the development of a Model-to-Model transformation that creates the Model-of-View (the Template-of-View in the MDA approach).

Listing 5.10 contains an example for such a transformation using the Metamodel generated from the VLD example of the previous section (see Listing 5.7 and Figure 5.12). It clearly shows how easy it is and how little information needs to be provided to create a Model-of-View (Line 10) using some input information (Lines 3-7). Once the input information is available, a single line of code creates the Model-of-View and any other steps are fully automated.

Listing 5.11 shows the equivalent view generation using a traditional, template-based approach, without a Template-of-View and without the VLD-based generation of a Model-of-View and automated view generation. The Mako-based approach starts with the same pre-defined input information (Lines 3-7). Instead of a single line generating the MoV as shown in Listing 5.7, the Mako approach requires lines 9 to 25 to generate the same target view. These lines take care of all the required steps, such as tabular alignment and indentation manually. It is for example first necessary to iterate over the `Name` and `Mode` attributes of all ports to find their maximum width. This information is later used in Python formatting directives (e.g. `"{s: <{w}}".format(s=port["Name"], w=max_name_width)`) to properly pad the variable contents with whitespaces.

```
# data derived by the Model-to-Model transformation, e.g. by iterating
# over MetaRTL and mapping to the semantics of the target HDL.
entity_name = 'sample'
vhdl_ports = [{'Name': 'short_port_name_i', 'Mode': 'in',
               'Type': 'std_logic_vector(31 downto 0)'},
              {'Name': 'very_long_port_name_o', 'Mode': 'out',
               'Type': 'std_logic'}]

# generation of the Model-of-View
model_of_view = Entity(Name=entity_name, Ports=Ports(Ports=vhdl_ports))
```

Listing 5.10: Template-of-View Excerpt to Visualize the Benefit of VLD-based View Generation

```
<% # data derived by the Model-to-Model transformation, e.g. by iterating
   # over MetaRTL and mapping to the semantics of the target HDL.
entity_name = 'sample'
vhdl_ports = [{'Name': 'short_port_name_i', 'Mode': 'in',
               'Type': 'std_logic_vector(31 downto 0)'},
              {'Name': 'very_long_port_name_o', 'Mode': 'out',
               'Type': 'std_logic'}]

max_name_width = 0
max_mode_width = 0
for port in vhdl_ports:
    max_name_width = max(max_name_width, port["Name"])
    max_mode_width = max(max_mode_width, port["Mode"])
%>
ENTITY ${entity_name} IS
% if len(vhdl_ports) != 0:
        PORT(
%     for port in vhdl_ports:
        ${"{s: <{w}}".format(s=port["Name"], w=max_name_width)}\
 : ${"{s: <{w}}".format(s=port["Mode"], w=max_mode_width)} ${port["Type"]}\
${'' if loop.last else ';'}
%     endfor
        );
% endif
END ${entity_name};
```

Listing 5.11: View Generation based on the Mako template engine [98] using input equivalent to Listing 5.10

The issues that were identified with Template-based code generation are now all addressed with the VLD-based view generation:

Syntactical correctness is guaranteed by a correct View Language Description. The VLD generates the Metamodel. The data validation performed on the model as part of the generated API ensures that all necessary constraints are enforced.

Formatting Human readable target views with correct formatting, indentation, and alignment are automatically generated as described in the VLD.

Configurability Different formatting and coding styles can be supported with this approach in multiple ways:

- Any changes to VLD files that do not affect the attributes, rules, and associations (i.e. all changes to the VLD which are limited to terminal symbols and formatting rules) can be performed to generate differently formatted views without changing the Metamodel-of-View and therefore without affecting the transformations.

- Model-to-Model transformations on the Model-of-View can be used to change the target view after it has been generated.

An additional significant benefit of the approach is that the Model-to-Model transformation which replaces the template-based view generation is entirely in the environment of the Metamodeling framework's programming language. This guarantees excellent tool support and syntax highlighting which further eases the development process. In the Model Driven Architecture framework established in this thesis, the VLD-based generation is used to generate VHDL views from the Model-of-Design representation. This generation of views is reduced to mapping the structural MetaRTL MoD, which is exclusively based on hardware semantics onto the VHDL hardware description language with its simulation semantics.

This simplification is an important step to enabling more powerful generators that are also applicable outside the Model Driven Architecture framework presented here. Depending on the Metamodel used for a view generation task, the challenge of generating the view is complex enough and does not need to be further complicated by the challenges listed here. This challenge is even larger in absence of a multi-layer Model Driven Architecture flow as the input gap in abstraction that needs to be bridged between the input models and the target view is larger.

6. Application and Subsequent Work

This thesis has so far introduced the adaptation of Model Driven Architecture for Digital Design and has demonstrated how it is implemented on top of Infineon's Metamodeling Framework. The feasibility and benefits of the approach was demonstrated by designing different versions of a CPU core that implements the RISC-V instruction set architecture with a microarchitecture using a five-stage in-order pipeline.

6.1. Application of Generators in Commercial Designs

In the time span between the start of this thesis work and its completion, the MDA framework has laid the foundation for the MetaX initiative at Infineon Technologies AG. The goal of this initiative is to use the Model Driven Architecture vision and its adaptation to digital design in commercial projects. In the context of this initiative, many designs have been realized based on the MDA framework presented in this thesis.

A key asset of the MetaX initiative is an SoC generator that makes use of a large collection of MetaX generator IPs. A RISC-V CPU core with a five-stage pipeline which was developed in the context of this thesis is one of the generator IPs part of this collection. This core was subsequently extended with additional features such as debugging support and selective hardening. Other CPU architectures such as an FSM-based single-stage architecture, as well as a two- and three-stage pipeline version of the CPU generator, are available as well. All these variants rely on the same Model-of-Things called *MetaRISC*. This Metamodel provides configurability of each of the CPU cores, allowing to enable or disable instruction set classes and instructions as well as countless other generation properties. These different implementations can therefore be used interchangeably with the same Model-of-Things configurations, providing implementation alternatives for architecture exploration.

Figure 6.1 shows a sample SoC that can be generated and configured by the MetaX generator framework developed in this thesis. In addition to the CPU cores with their architectural alternatives, MetaX provides generators for various peripherals. These peripheral generators include general-purpose timer IPs, communication IPs (e.g. UART, SPI, GPIO IPs), debugging interfaces for JTAG, and accelerator IPs (e.g. for CRC calculation). Moreover, it covers infrastructure generators for different bus protocols (a proprietary internal bus protocol, AHB and APB) and a generator for interrupt controllers. In addition to the individual generator IPs, an SoC generator is available to perform the toplevel connectivity and instantiation of infrastructure resources.

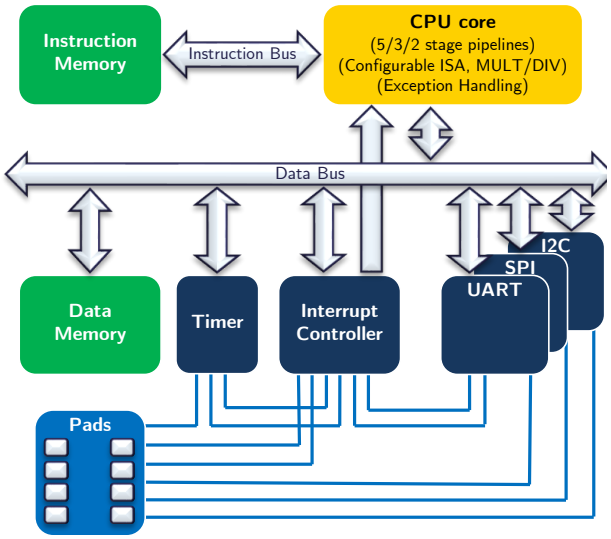

Figure 6.1.: Sample SoC generated with MetaX generator framework

All available generators follow the same high-level structure defined by this thesis: an abstract Model-of-Things Metamodel is developed that is specific to the function of the peripheral. For a communication peripheral, this Model-of-Things would define generation options such as the number of channels supported by the peripheral, the bandwidths and protocols supported by these channels, and use-case and integration-specific options such as the sizes of buffers inside of the communication peripheral.

the MDA approach has demonstrated its capability to design efficient hardware generators in its use for the development of these MetaX framework IPs and in their application in an industrial context at Infineon Technologies AG. Using the MetaX generator framework as a replacement for traditional manual coding has shown a reduction of effort of up to 50% compared to manual implementation and adaptation without code generators. The effort needed with the generation approach also includes the development of generators.

The MetaX framework is set to be integrated into the Infineon design flow. In this environment, it can be commercially applied by chip design teams to different extents: the integration for example supports the development of individual IPs as part of an overall system designed with traditional hardware description languages. This is particularly well suited for highly dynamic and configurable IP components such as bus fabrics or I/O subsystems that provide a configurable mapping from physical IC pins to internal IP functions. For this use case, the MDA framework has been extended to also generate collaterals needed for SoC integration from the Model-of-Design. Using MetaRTL in this context does therefore provide benefits: a significantly higher level of configurability and reduced effort for SoC integration.

MetaRTL can also be used in a wider scope in Infineon's design flow. So far, it has been utilized to generate entire digital designs for test chips and for entire design subsystems for ASIC development at Infineon Technologies AG.

6.2. Architecture Exploration using MetaX

A significant recent development that is used for research at Infineon Technologies AG is the emergence of the OpenROAD and OpenLane open-source EDA flows. In recent years, these flows have become a viable and useful addition to commercial, proprietary EDA tools and flows. To date, commercial EDA tools still have a significant qualitative edge over their open-source competitors and cannot be replaced for scenarios where production quality results are required. The available open-source tools and flows are however for areas where a large number of tool runs are needed and the tool outcome is not used for production but to make architectural decisions. At Infineon Technologies AG, the OpenROAD and OpenLane frameworks are used together with techniques of the Machine Learning (ML) domain and for example for parameter exploration and tuning for physical synthesis. [75, 81, 90]

In a common use case, OpenROAD's AutoTuner is utilized to automatically find optimal parameters for the physical implementation flow. These parameters can then be used with commercial tools for later work on productive tapeouts. The Model Driven Architecture framework and the full chip generators developed as part of the MetaX are a valuable addition to this use case. Without MetaX, the typical scope of the hyperparameter exploration is the entire RTL-to-GDS flow, starting with synthesizable, elaborated, and parameterized RTL views. With the help of MetaX, the scope can be extended significantly: for each of the generators available as a part of MetaX, the generation parameters provided by its Model-of-Things can be added to the exploration. This allows for example to automatically generate a different number of pipeline stages for the CPU core and evaluate the power, area, and performance impact of this change together with changes in other parameters of the RTL-to-GDS flow. [90]

6.3. Subsequent Work

The MDA framework presented in this thesis has however also been extended beyond its initial capabilities and scope within digital design. Figure 6.2 shows the extensions made to the MDA framework. The most noteworthy are the extensions beyond the scope of digital design for formal verification and firmware development. The extensions into these areas are the outcomes of two separate PhD theses [85, 97]. These theses reuse and adapt the concepts, framework, and tooling provided by this thesis and transfer them to the respective domain.

- All approaches use the same three-layer structure defined in this thesis. Along with this, they share an important aspect of the generation framework as they are able to utilize the same Model-of-Things inputs. In other words, they can all access the same common specification inputs and utilize them to generate target views of the digital design, of verification properties for the design, and of firmware code that can be executed on the design. This generation from a shared common formalized specification layer ensures consistency between the separate target views.

- All approaches use the method for Model-to-Model transformations defined and developed

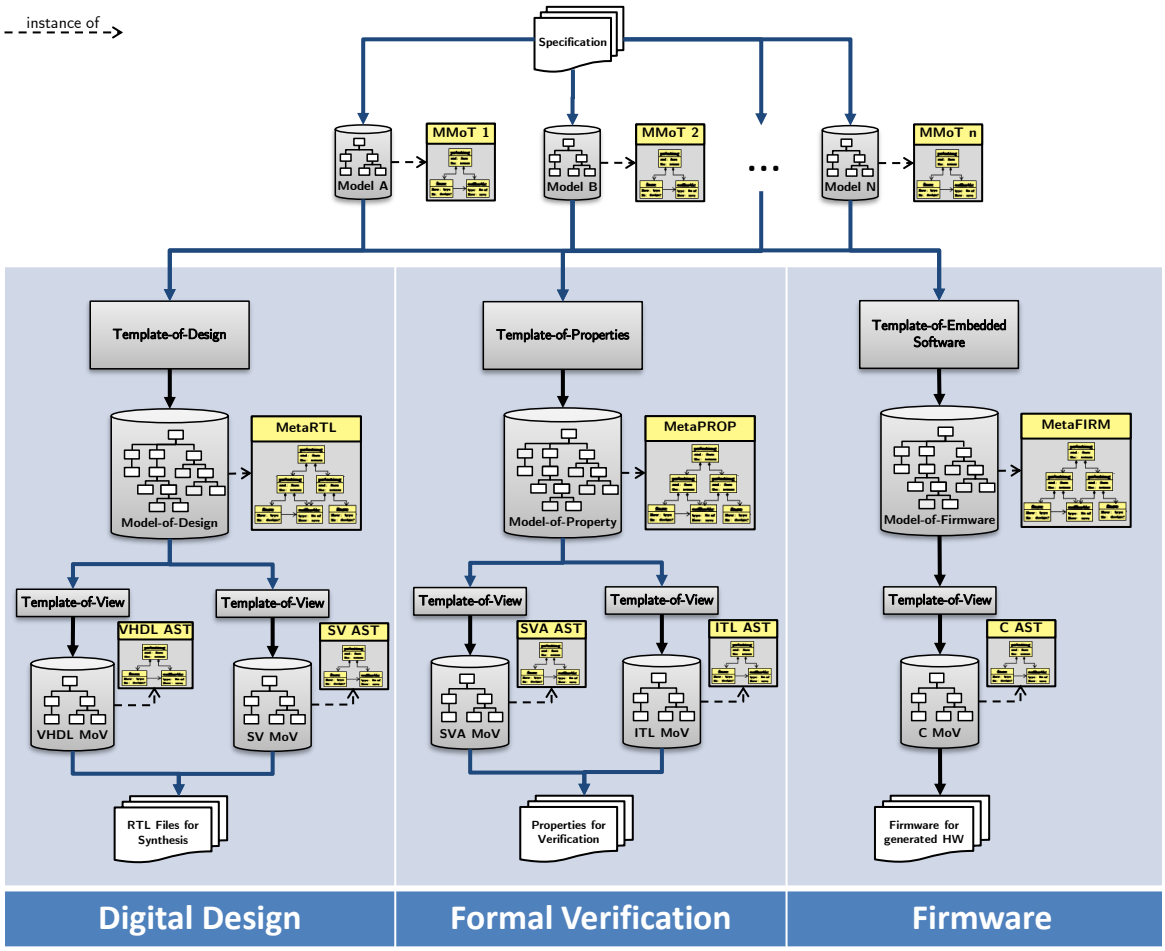

Figure 6.2.: Extensions of the MDA framework beyond Digital Design

for the Template-of-Design and Template-of-View in this thesis: they are using Python and extended Metamodeling APIs to access the specification Metamodels and generate different target Metamodels from them. Similar to this thesis, they extend the generated APIs with domain-specific APIs to provide significant reductions in development effort and verbosity for their transformations [85, 97].

- All approaches use the same VLD method and unparse tooling developed as part of this thesis: they each add additional target view languages to the framework: the System Verilog Assertion (SVA) language and Onespin's ITL language for formal verification and the C language for firmware generation. In addition to that, the MDA flow for digital design was extended with support for System Verilog views as part of an internship project. [85, 97]

The generation flow for digital design introduced in this thesis defines MetaRTL, the Metamodel of the Model-of-Design layer which captures a structural representation of the design with synthesis semantics. Both the additions for formal verification and firmware development

define separate Metamodels that act as the Model-of-Design level counterpart to MetaRTL. The MDA framework developed in this thesis only covers digital design and is centered around the central Metamodel MetaRTL, the overall framework that was developed subsequently also includes separate Metamodels as counterparts to MetaRTL for formal verification and firmware development. This extension in scope also explains the name derived for this overall framework *MetaX*, representing the MDA flows for digital design (centered around *MetaRTL*), for formal verification (centered around *MetaPROP*), and for firmware development (centered around *MetaFIRM*).

For formal verification, MetaPROP is developed to capture the temporal semantics of design properties. An additional Metamodel called Model-of-Binding which is not displayed in the figure adds essential information to the properties defined by MetaPROP instances. The Model-of-Binding references the aspects of the MetaRTL Model-of-Design that a design property applies to. This linking is used as part of the Template-of-View transformation to generate assertions in the target languages which are linked against the design elements in a MetaRTL model that the assertion applies to. [85]

For firmware development, the MetaFIRM Metamodel is the counterpart of MetaRTL for software development. It captures the semantics of sequential software execution. As such, it contains a description of the state internal to the software (variable declaration) and a definition of individual software blocks (function declarations) as well as software interfaces: the Application Programming Interfaces (APIs) that individual software components exhibit to their higher- or lower-level components. In addition to that, MetaFIRM models the control- and dataflow of the individual functions using a semantic similar to UML activity diagrams with control nodes (for loops and conditionals) or data nodes (for actual computations changing the software's internal state). [97]

The extension of the proposed MDA flow to areas beyond the scope of Digital Design highlights a key advantage a Model Driven Architecture based approach has compared to approaches relying on Embedded DSLs without the backing of formally defined Metamodels (e.g. Chisel). The MDA approach presented here relies on Metamodeling and an underlying Metamodeling Framework. This Metamodeling-based approach introduces many commonalities in the different MDA flows for digital design, formal verification, and firmware development. From the perspective of a developer applying MetaX for generator development, this homogeneity reduces the learning needed to get familiar with all three generation flows. All transformations that are needed (Template-of-Design, Template-of-Property, and Template-of-Embedded-Software) are developed using the same structure and the same automatically generated API (defined by visual Metamodels of the source and target models). The knowledge on how to develop working code for one defined Metamodel (e.g. the Template-of-Design) can therefore easily be transferred to other Metamodels (e.g. the Template-of-Property). Purely language-based approaches that do not make use of an underlying Metamodeling framework require the user to learn the semantics of the language itself, a task that is significantly more difficult than understanding the source and target Metamodels.

Moreover, the ability to cover both firmware and hardware development as well as hardware verification in one single holistic environment guarantees consistency between the domains. All

changes to the Model-of-Things input can automatically propagate into all domains. For any change in input data, the flow itself can regenerate the affected target views. This for example guarantees that potential additional assertions are generated and software is automatically adjusted to reflect hardware changes. [85, 97]

7. Generator IP-reuse and Automated Infrastructure Generation

The application of MetaX in internal research and for productive designs has already demonstrated its ability to significantly reduce NRE efforts. It has also helped to identify two key limitations for the development and application of reusable generators.

1. Generators need to be developed with the individual SoC integration aspects in mind. This means they have to be developed using one or more assumed SoC ecosystems into which they will later be integrated. Supporting different system-level requirements such as different bus protocols, different memory architectures, or interrupt protocols often requires modification of individual generators and prevents clean reuse.

   In the MetaX generator library, this problem becomes particularly apparent when individual generators are used to generate IPs that are later used outside of the context of the MetaX SoC generator as individual generated instances inside a project that does not use the MDA approach for its toplevel. Here, the architecture and infrastructure of the SoC may differ from what was assumed during the IP generator's development and may require changes to the IP generator to achieve compatibility.

   It is of course easily possible to modify the IP generator to support a different integration context (e.g. by providing an additionally supported bus slave interface as an additional generation mode). Such a modification of the generator must undergo rigid testing to guarantee it is backward-compatible and that it does not break any existing support for other SoC contexts. It is however still a modification of an existing IP generator, prohibiting its full reuse. Moreover, this modification adds additional complexity and SoC integration details to the IP generator of the modified IP.

2. IP integration aspects need to be taken care of by every component's parent component, or the SoC generator. Providing the right kind of SoC infrastructure (e.g. right bus architecture, right interrupt routing, or right memory test architecture requires configuration of the SoC generators that is compatible with the microarchitecture generated by the IP generator).

   In the MDA approach, all configuration of generator components happens based on Model-of-Things instances and is constrained by the Metamodel-of-Things of the individual generators. It is therefore necessary to adapt the Model-of-Things instances of the infrastructure and SoC components to provide compatibility with the resource needs of the utilized generator IPs. In other words, all Model-of-Things instances for all generators need to be compatible.

Part of this work cannot be automated and is fundamental manual work in the MDA flow: providing the Model-of-Things as a formalized specification input to the generator is what replaces the task of manual coding. A significant part of this work should however be automated: the generation of Model-of-Things information for infrastructure components. The key reason for this is that information on the infrastructure needs of any individual generated IP is known and understood by the Template-of-Design (i.e. the program code) of the IP generator. At the point where the IP generator's program code is executed, the generator determines a microarchitecture to implement the Model-of-Things. At this point, the generator is aware of the infrastructure needs of the generated hardware.

Automating these aspects is particularly important in a generator context. Here, different microarchitectures generated for the same Model-of-Things descriptions may require different infrastructure needs and a manual adaptation of Model-of-Things instances may not be possible when the architecture is automatically generated.

This chapter introduces a solution to these problems that is based on a generator architecture using best-practice software architectural patterns that allow generators *to communicate and collaborate to align on a common, optimal architecture.* The solution is centered around a generator architecture based on best-practice software architectural patterns. The proposed architecture has two key advantages:

1. IP generators can be developed entirely independently from SoC integration aspects. The provisioning of SoC integration inside the generated IPs (e.g. optimal bus interface sub-components) is transparently delegated to specialized generators. Changing these centralized generators and along with that generated hardware as well as utilized protocols can happen without modifying the IP generators intended for reuse.

2. Infrastructure generators automatically collect information on the infrastructure needs of the system that is generated by the individual IP generators. Using this information, the infrastructure generators can automatically and transparently generate suitable SoC infrastructure (e.g. bus fabric with topology adjusted to the individual generators) and provide the toplevel connectivity for their infrastructure.

In the following, this chapter describes the two limitations listed above in more detail and with examples. It then analyzes the structure of the generators part of the MetaX initiative to identify the root cause of the limitations. Based on this analysis, the so-called *Multi-Pass Generation* is proposed and described as a new pattern for ToD development. This pattern removes the limitations identified.

7.1. Detailed Problem Statement

The application of MetaRTL in internal research and for productive designs has already demonstrated its ability to significantly reduce NRE efforts. It has also helped to identify two key

limiting factors for the development and application of reusable generators. These limitations are shared by the Model-based generation approach and popular Hardware Generation Languages such as SpinalHDL and Chisel.

7.1.1. Limitation 1: Decentralized Handling of Infrastructure Needs

IPs Generators need to be developed with the individual SoC integration aspects in mind. In comparison to the traditional development of static digital designs in Hardware Description Languages such as SystemVerilog and VHDL, IP generators provide significantly more flexibility. While traditional IPs often need to be wrapped or modified to provide compatibility with a certain SoC, an IP generator can be developed to support multiple different SoC ecosystems. The main limitation here is that these SoC ecosystems have to be known and considered when the generator is developed. Integrating a generated IP into a system that for example uses a bus protocol, a memory architecture, or interrupt protocols that are not supported by the generator still requires modification of the generator code and thus prevents clean reuse.

This limitation is particularly evident when utilizing the generator library to generate IPs that are integrated into productive SoC designs that use a traditional, HDL-based design flow. Here, the architecture and infrastructure of the SoC often differ from what was assumed during the IP generator's development and changes to the IP generator are required to achieve compatibility. It is of course possible to modify the IP generator to support an additional integration context. Such a modification of the generator must however undergo rigid testing to guarantee backward-compatibility and does not break any existing support for other SoC contexts. Moreover, this modification adds additional complexity and SoC integration details to the IP generator of the modified IP.

It is straightforward to find examples of changes required by different SoC contexts:

- A component that acts as a bus slave or master must be generated with the correct design interface (ports) to connect to the bus fabric. It must moreover instantiate the necessary slave or master interface, supporting the exact bus protocol. More complex requirements such as heterogeneous bus architectures may also force individual components to come with multiple bus interfaces in one SoC while for another SoC, a single, general-purpose interface may be sufficient.

- A component that requires volatile memories (such as cache memories in CPUs or buffer memories in communication peripherals) needs to support the structure used by the SoC for local memories.

  Some SoCs for example instantiate all memories on top-level (memory on top), outside of the modules that use them. When an IP generator is integrated into such an SoC, the module has to be generated with additional ports to access the read and write ports of these memories.

In other SoCs, the memories may be instantiated inside the individual modules that make use of them (distributed memory). In this case, the generator has to instantiate the memory sub-component internally. In addition to using the read and write interfaces of these instantiated memories, additional connectivity may need to be provided to the memory module instantiated as a sub-component. For example, automotive applications come with high safety and reliability standards and memories used in these applications have stringent memory test requirements. This can mean that the memories need to be wrapped with test logic or connected to centralized memory test units which control features such as power-on self-tests of memories.

Throughout this chapter, the example of memories and memory test is used because of its simplicity. The main benefits of the work published here is however in applying the methodology to all kinds of infrastructure resources instead of limiting it to memories. Figure 7.1 contains an excerpt from a few sample SoCs with different memory architectures. It clearly shows how these impact the IPs. In examples a) and b), the individual IPs X and Y need the connectivity for memory test interfaces (yellow) and need to instantiate the memory modules. In example c), the individual IPs X and Y will only need to connect the read and write ports (green) to their respective boundaries. Similar differences become apparent on subsystem and toplevel. In example a), the subsystem needs to contain interfaces and connectivity for the memory test, in example b) it needs to contain connectivity and to instantiate a distributed memory test unit, whereas in example c), it needs to contain read and write ports and connectivity for them.

We already mentioned that modifications to IP generators are not considered as clean reuse and come with stringent verification requirements. With the given example, it also becomes clear that IP generators need to be often modified although they are not functionally affected: In a distributed memory architecture, it may be required to add an additional test wrapper around the memory. As long as the read and write interface of the memory is not modified by the memory wrapper, this additional layer is transparent and does not require changes to the core functionality of the IP instantiating the memory. Nevertheless, a modification will require the modification of the parent module: it needs to add additional ports to the generated module to connect the centralized memory test controller to the memory wrapper.

From a point of view of the software architecture, this is not desirable. The software architecture of the generator violates two of the five popular and widely accepted design principles of object-oriented design summarized with the mnemonic SOLID.

- The first violated design principle is the *Single Responsiblity Principle* (the S in SOLID). This principle states that "[...] each software module has one, and only one, reason to change" [72]. A logical consequence of this observation is that a class should only have one job [24]. The parent module should be responsible only for providing its function and not for providing the potential infrastructure functionality of its sub-modules.

- The second violated design principle is the *Interface Segregation Principle* (the I in SOLID). "This principle advises software designers to avoid depending on [software in-

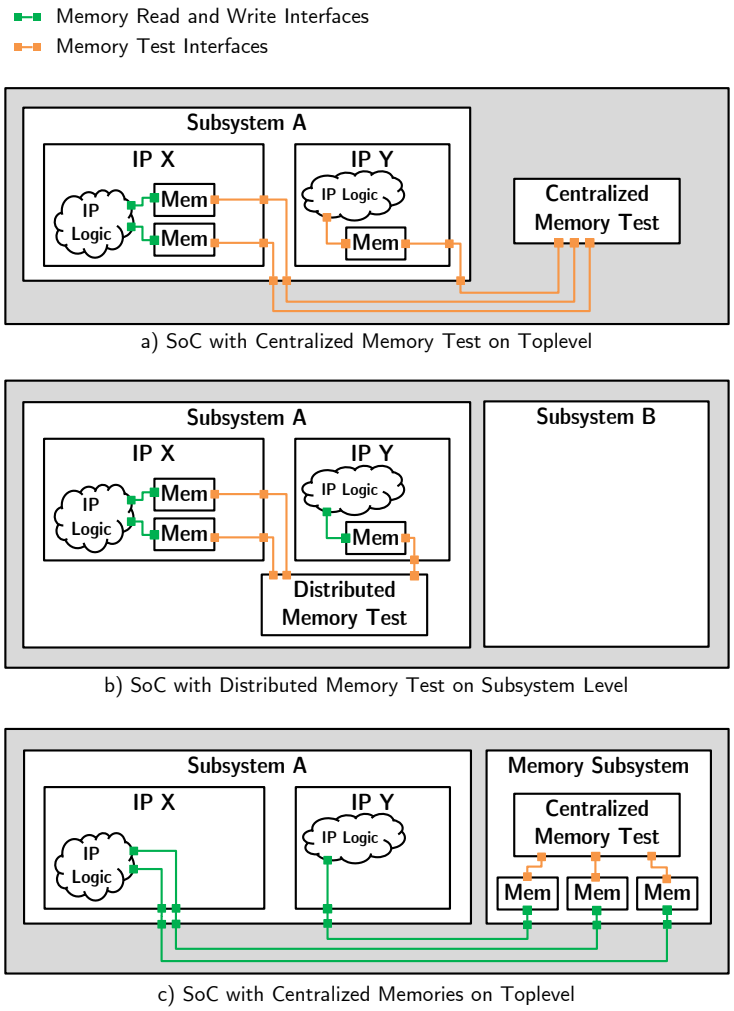

Figure 7.1.: Example for different Memory Architecures

terfaces[1]] that they don't use" [72]. In this scenario, the generator of the parent module is using the generator for the child module (it is using the memory generator to get the required memory module). In the context of this usage, the required infrastructure for the module provided by the sub-generator is explicitly not an interface that it depends on (it depends only on the read and write functionality provided by the module). For situations such as this, the Interface Segregation Principle states that the parent generator should not depend on or use an interface it does not need.

[1]So far, the term interface is used to refer to *design interface*, i.e. the ports of a design component part of an RTL design. Here, the term interface refers to the concept of *software interface*. Software interfaces are used in object-oriented programming. They define a set of methods with their arguments and return types that a class must implement if it adheres to the defined interface. The interface acts as a contract that describes what can be done with an object of a type that implements the interface. The most important difference between base classes and interfaces is that interfaces must not contain implementations.

7.1.2. Limitation 2: Provisioning and Configuration of Infrastructure Resources

We have demonstrated that in many cases, generators of modules need to add additional ports and internal connectivity to the generated module to account for the infrastructure needs of sub-modules. This causes additional problems when further stepping up the hierarchies. Eventually, it is not sufficient to just connect the ports required by the sub-modules to the parent module hierarchy. Depending on the overall architecture, the infrastructure requirements have to be met on some hierarchy, either on subsystem level or on SoC toplevel.

In an environment with multiple different generators which have their own microarchitecture options and configurability, highly variable needs on centralized resources can arise. It is equally straightforward to come up with examples here: The amount of SRAMs that are connected per memory test controller depends on the amount of read and write bits that each individual SRAM has as the number of overall connections to the test unit is limited by routing constraints. Different SRAM types and performance characteristics may require different memory test controllers. Depending on the safety and reliability requirements for the attached SRAMs, the algorithms a memory test controller has to be able to execute may differ: while power-on self-test is sufficient for most applications of on-chip SRAM, some special applications may require non-destructive test at runtime (for example to provide long uptimes in industrial applications where equipment reboots are not feasible).

Similar to all the generators, the generators for infrastructure components come with their own Metamodel-of-Things, describing the configuration that is possible for the generator. This Model-of-Things needs to be adjusted to ensure the generated infrastructure meets the exact requirements of the instantiated IP components of a given SoC. This is problematic as the Model-of-Things in the flow is the layer of formalized specification that is typically manually entered by users. The need to provide matching Model-of-Things specifications for the individual components makes the application and execution of generators more complex, often requiring detailed knowledge about the properties of the generated microarchitectures.

In an industrial context with traditional HDL-based development of digital designs, this issue also exists. Solving it requires complex coordination between different teams responsible for different sub-components. Typically, centralized teams collect information about the infrastructure needs of a design and ensure that central modules which are compatible with the SoC needs of all components are provided.

The generation-based automation can already fully automate the generation of centralized infrastructure components when these infrastructure needs are formalized as Model-of-Things instances. What the current approach can however not do is automate the collection of the required information. The use of flexible generator-based approaches to IC design only exacerbates this issue. What kind of SoC resources a sub-component needs depends highly on the Model-of-Things that was used to generate it and on the microarchitectural decisions made as part of the Model-to-Model transformation. It is straightforward to come up with examples: A communication peripheral may or may not use an SRAM to buffer the I/O data. The frequency and capacity of the SRAM may depend on the throughput required and defined

in the Model-of-Things. The number of SRAMs may depend on the number of channels the communication peripheral is using. The number of SRAMs that are needed may depend on the microarchitectural properties of the I/O pipeline of the peripheral (how many channels are served per I/O pipeline). The safety and reliability needs of the data stored in the SRAM may influence the requirements on memory test circuitry that needs to be provided to the SRAM. For some applications of on-chip SRAM, the integrity of the data stored will always be protected by other mechanisms (for example a higher-level protocol's CRC). In other applications, the SRAM is used to store data which has no further protection mechanism and the SRAM, therefore, requires extensive hardware protection mechanisms.

This limitation is particularly critical for one area of the research goal: Automatically assess the capabilities of different microarchitectural alternatives provided by the generators. For this purpose, hyperparameter exploration algorithms can be applied to automatically modify the configuration of the RTL generators, resulting in different generated architectures on Register Transfer Level. In addition to changing characteristics such as the power, area, and performance of the generated hardware, these parameter modifications often impact the required SoC infrastructure and it is challenging to align the different Models-of-Things used for one SoC when parameters are automatically altered. The generation of suitable common infrastructure thus needs to be automated. [90]

7.2. Root Cause Analysis

The limitations identified in the previous section are primarily caused by the current best practice used for generator development. Section 7.2.1 describes this current best practice, referred to as *Single-Pass Generation* in this work. This pattern is strictly enforced by VHDL and SystemVerilog. In contrast, it is not enforced by Hardware Generation Languages such as Chisel and SpinalHDL or by the Model-based Template-of-Design approach. Section 7.2.2 explains the rationale for why this pattern is still followed in HGLs and in the Templates-of-Design libraries. Section 7.2.3 suggests an alternative pattern that solves the limitations identified.

7.2.1. The Single-Pass Generator Pattern

As part of the Template-of-Design, every IP generator is typically implemented as its own Python class which inherits from the `Structure` class part of the MetaRTL Metamodel-of-Design. The FIR filter generator described in Listing 5.6 is an example of such a reusable Template-of-Design class. This class takes Model-of-Things instances as its constructor argument. When the class is instantiated, the constructor is called, inserts the structure into the Model-of-Design, and fills it with functionality. During the execution of the constructor, the Model-of-Things is analyzed and the structure is built. The moment the constructor returns, the Model-of-Design contains all elements describing the static IP instance as MetaRTL structure: all required connectivity, logic, sub-components, and – most importantly – all required

inputs and output ports of the generated structure. This concept is what is referred to as *single-pass* ToD execution. The key observation about *single-pass* ToD execution is: the moment any constructor finishes its work and returns an instance of a Model-of-Design `Structure`, it is completely statically built and all its properties are well defined.

When generators are implemented using this single-pass approach, the sub-components and logic that the constructor generates inside the Model-of-Design may either be blackboxes provided by MetaRTL (e.g. `Register` or `Primitive` components) or – following the concept of hierarchical design – their own complex structures built by child generators, which are in turn specializations of (i.e. classes derived from) the MetaRTL `Structure`. Here, the constructor of the parent structure will in turn instantiate sub-structures provided by other generators (each specialization of the MetaRTL `Structure`) by calling their sub-constructors.

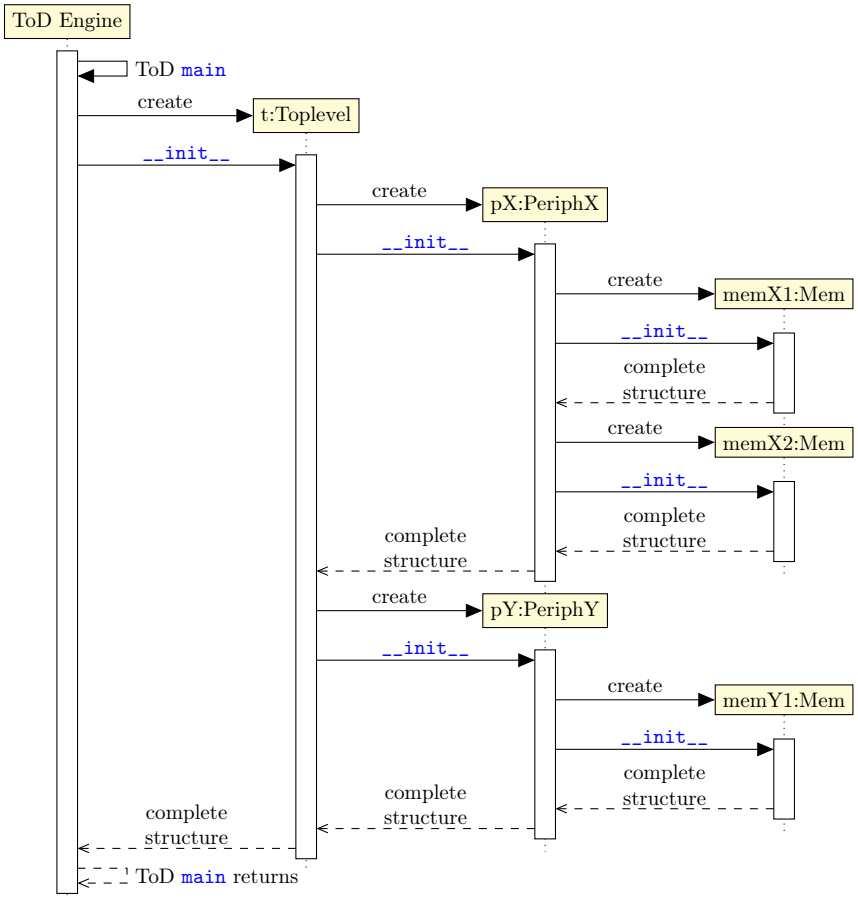

Figure 7.2.: UML Sequence Diagram of Template-of-Design in Listing 7.1 with single-pass creation of nested structures

Listing 7.1 contains a simplified skeleton of a Template-of-Design with multiple nested generators that highlights the order in which the nested structures a built. The `main` function in line 33 is the entry point for the Template-of-Design. Starting from here, every `__init__` is called once and returns a complete structure, including all its substructures, connectivity,

```
1  class PeriphX(Structure):
2      def __init__(self, parent=None):
3          super().__init__(Ports=[ ... ],
4                           parent=parent)
5          ...
6          memX1 = Mem(self)
7          memX2 = Mem(self)
8          ...
9          # end of constructor,
10         # Structure fully finalized
11
12 class PeriphY(Structure):
13     def __init__(self, parent=None):
14         super().__init__(Ports=[ ... ],
15                          parent=parent)
16         ...
17         memY1 = Mem(self)
18         ...
19         # end of constructor,
20         # Structure fully finalized
21
22 class Toplevel(Structure):
23     def __init__(self, parent=None):
24         super().__init__(Ports=[ ... ],
25                          parent=parent)
26         ...
27         pX = PeriphX(self)
28         pY = PeriphY(self)
29         ...
30         # end of constructor,
31         # Structure fully finalized
32
33 def main(Api, MoD):
34     t = Toplevel(parent=MoD)
```

Listing 7.1: Sample Template-of-Design with single-pass creation of nested structures

and interfaces. The sequence of calls and the control flow through the application can be best visualized with the UML sequence diagram in Figure 7.2. Here, the dashed arrows show when a structure is finalized:

1. `memX1`, after the `__init__` called by the instantiation in line 6 returns
2. `memX2`, after the `__init__` called by the instantiation in line 7 returns
3. `pX`, after the `__init__` called by the instantiation in line 27 returns
4. `memY1`, after the `__init__` called by the instantiation in line 17 returns

5. `pY`, after the `__init__` called by the instantiation in line 28 returns
6. `t`, after the `__init__` called by the instantiation in line 34 returns

It is important to note that this pattern is not specific to the model-based approach: As mentioned in the introduction to this section, the same applies to traditional HDLs: the language construct that instantiates a module in SystemVerilog or a component in VHDL defines the exact parameterization of the design. Upon elaboration of a SystemVerilog or VHDL module, this line is what provides a fixed definition of the parameter set used for the instance. Moreover, the pattern is also applied in Chisel and SpinalHDL and equivalent examples can be derived in these hardware generation languages.

7.2.2. Benefits of Single-Pass Generation

While it is helpful to allow generators to build structures with changing interfaces, it is important that the moment the constructor returns, the defined design interfaces (module ports) are static. Dynamic modifications of both the interfaces and functionality after the generation of a module have shown to be particularly hard to handle: how should a parent generator make use of the functionality of the child generator if it does not know how to interface it? The same generally applies to the internals of any generated module. When changing the functionality after the parent generator has finished, who is responsible for modifying the structure generated by the parent generator? How is it possible to find out what needs to be done in a sub-structure without completely reimplementing the sub-generator's logic?

When the established pattern of Single-Pass Generation is followed, any parent structure knows how to connect its hardware to the child structure as it can make use of the complete and static design interface of the child structure once its constructor returns. The assessments in this area show that the simplicity of this approach and its high reliability is what make it the gold standard for hardware generation to date.

A static and well-defined design interface is also important for generator reuse. In the current best-practice approach, a generator invokes a certain sub-generator with a defined configuration as provided by a Model-of-Things instance handed to the constructor of the sub-generator. The parent generator has to rely on a contract that defines a design interface and functionality for the generated submodule for any given Model-of-Things instance. Any internal change of the sub-generator, for example, an extension of its functionality is not critical for the parent generator as long as this contract is not violated.

7.2.3. Multi-Pass Generation

To come up with a solution for the identified limitations, a closer assessment of the contract defined for any IP generator library was conducted. In general, this contract consists of several properties. These properties include the versions of the model-based generation framework the

generator is compatible with, the dependencies the generator has on other sub-generators (and which versions of those sub-generators it is compatible with), and the Metamodel-of-Things that have to be used to configure the generator.

In addition, the following properties are considered essential elements of this contract:

1. IP Functionality
   a) the functionality a component provides to the outside world
   b) the design interface (ports) a component exposes to use the functionality
2. Infrastructure Dependencies
   a) the infrastructure resources required by the component to provide its functionality
   b) the design interface (ports) and connectivity that the component requires to access the infrastructure functionality

In traditional Hardware Description Languages, these properties are mostly fixed (aside from minimal size configurability based on parameters, both the functionality, infrastructure requirements and interface of a SystemVerilog module or VHDL component are mostly static). In a generator-based ecosystem, these properties are not generally fixed. The MDA framework is suited for the development of IP generators that significantly modify the functionality ((1a) and (2a)) and even the design interface (ports) to the outside world ((1b) and (2b)) of a `Structure` that is being generated in the Model-of-Design. Nevertheless, the contract mandates that these properties are fixed for a given set of Model-of-Things input parameters (A different Model-of-Things can result in different functionality, infrastructure needs, or HW design interfaces).

The internal logic and interfaces built for each of the structures are those to access the functions provided by the structure ((1a) and (1b)) and those that are needed by the sub-structure to access required infrastructure components ((2a) and (2b)). It is important to note that (2a) and (2b) do not only include the infrastructure needed by the sub-structure themselves but also infrastructure that is needed by potential sub-sub-structures. For example, once the construction of `pX` in line 27 of Listing 7.1 returns, not only the infrastructure interfaces for infrastructure required by the peripheral itself need to be created. The constructed structure also needs to have all infrastructure interfaces and correct internal connectivity and logic for the infrastructure needs of its sub-structures `memX1` and `memX2`.

It has not been possible to identify a method that allows us to resolve the limitations identified while maintaining the established single-pass generation pattern. Upon closer inspection of the problem and the side effects the modifications to the Template-of-Design structure has on the overall design, the following was found: for any use of a generator, the provided functionality (1a) and provided interfaces (1b) need to be fully constructed and available when the constructor of the sub-generator returns. Only if this holds true, the generator of the parent structure can make use of the functionality of the sub-component that is instantiated by connecting it to its internal logic. A deviation from classical single-pass ToD construction for (1a) and (1b) is not meaningfully possible.

This limitation does however not hold apply to (2a) and (2b). From the point of view of any

parent generator, the functionality the sub-components require (2a) and the interfaces needed to provide them (2b) does not need to be completed to construct all the functionality of the parent structure.

What this means can be explained with a few examples using an arbitrary communication peripheral:

- It is sufficient to instantiate all SRAMs with only their read and write interfaces as needed by the peripheral. The interfaces for memory test are not required or used to construct the peripheral itself and can be provided later.

- It is sufficient to instantiate all HW/SW interface registers with a wrapper exhibiting their write, enable, and read ports without actually connecting them to a bus slave interface and bus interface logic. The peripheral can use this wrapper to receive information from the HW/SW interface or to provide information to it. Any logic to actually make them HW/SW accessible is not required or used to construct the peripheral itself and can be provided later. From the point of view of the peripheral, the sub-component that provides the bus connectivity is actually just providing HW/SW interface functionality: which and how many buses need to be connected to the peripheral is not a concern of the peripheral but an integration concern.

In all these examples the core task of the IP generator, the construction of the actual IP functionality and interface can be completed without the required infrastructure. This thesis, therefore, suggests a deviation from the established best practice pattern of single-pass generation. While a generation of (1a) and (1b) in an unchanged, constructor-based manner is still required, this requirement can be relaxed for (2a) and (2b). This means that infrastructure generation shall be omitted in the first pass of all generators. Multiple later passes can then extend the existing MoD structures, resulting in an overall flow that is referred to as *Multi-Pass Generation*.

7.3. Implementation of Multi-Pass Generation

This section describes the design pattern derived for multi-pass generator development. To ensure the Template-of-Design execution ends with correct and working generators, it needs to be guaranteed that all infrastructure resources will eventually be inserted into the existing structures and that this happens without unintended modifications and side effects. To provide these guarantees, a reliable library needs to be developed which follows intuitive design patterns for developing the multi-pass generators. A key goal is to come up with an architecture that addresses the limitation identified and does not violate fundamental software architectural principles. Here, the guidelines provided by the five popular and widely accepted design principles of object-oriented design summarized with the mnemonic SOLID can be applied. In the following, this section lists the complete set of SOLID principles and shows which objectives were derived from them for the implementation of the multi-pass pattern.

- The Single Responsiblity Principle. This principle states that "[...] each software module has one, and only one, reason to change" [72]. A logical consequence of this observation is that a module (i.e. class) should only have one job [24]. From this principle, the need for a clear and clean separation of concerns in the development of the generators is derived: every generator should generate what it is responsible for, request its dependencies, and delegate all tasks it is not responsible for. In the concrete example, IP generators should not have to take care of infrastructure requirements they have themself or their sub-components have.

- The Open-Closed Principle. This principle states that in order to build a software system that is easy to change and extend, they need to be split into modules that "[...] must be designed to allow the behavior of those systems to be changed by adding new code, rather than changing existing code." [72]. This principle shows that it must be possible to change and extend the infrastructure capabilities of the generators without changing existing generator code that just uses this infrastructure or just uses components that make use of this infrastructure.

- The Liskov Substitution Principle. This principle states that "[...] to build software systems from interchangeable parts, those parts must adhere to a contract that allows those parts to be substituted one for another" [72]. This principle emphasizes that a clean contract needs to be defined between generators. This contract should allow every generator to make use of sub-components and to make use of infrastructure components without contributing to providing or connecting the functionality of these infrastructure components.

- The Interface Segregation Principle. "This principle advises software designers to avoid depending on things that they don't use" [72]. This principle shows that the generators should only depend on the functionality provided by the sub-generators, not on the infrastructure requirement of these sub-generators. When this principle is followed, the generators will not contribute to depend on any interfaces related to infrastructure functionality required by their sub-generators and therefore be indifferent to any changes in the infrastructure functionality.

- The Dependency Inversion Principle. "The code that implements high-level policy should not depend on the code that implements low-level details" [72]. This principle refers to what is commonly implemented by Inversion of Control in software engineering. A key part of this principle is that higher-level software components do not implicitly make use of lower-level software components. Instead, the higher-level components depend on software interfaces only and can be configured with respect to which lower-level components they use. This principle highlights that it needs to be possible to control which infrastructure providers are used when a parent module with a certain sub-generator is instantiated.

7.3.1. Definition of Interfaces for Communication between Generators

A key part of the suggested approach to generator development is the definition of reusable software interfaces that are used to communicate between generators. These interfaces are a core element of realizing the Interface Segregation Principle and the Liskov Substitution Principle. It defines a contract between two generators. Any other generator that also adheres to this contract can be used interchangeably as part of the component. Using this principle, infrastructure generators in the generator libraries can simply be replaced without touching the generators depending on the infrastructure generators.

```
1  from abc import ABC, abstractmethod
2
3  class IMemoryGenerator(ABC):
4      @abstractmethod
5      def getMem(self, parent, memoryMoT) -> None:
6          """
7          Generate a Memory and insert it into MoD
8          :param parent: the MoD Structure into which the memory is inserted
9          :param memoryMoT: the MoT model describing the memory properties
10         """
11         pass
```

Listing 7.2: Definition of Memory Generator Interface

Listing 7.2 shows a simplified example of such a generic interface called `IMemoryGenerator` that all memory generators have to implement. The Python language itself does not support software interfaces as a separate construct. To define an interface in Python, an abstract base class (abstract meaning it cannot be instantiated) that defines methods but does not include any method behavior is utilized. Any Python class that implements this interface simply inherits from the abstract base class and overrides the abstract methods.

7.3.2. Application of the Dependency Inversion Principle

Based on the definition of interfaces for infrastructure generators suggested in Section 7.3.1, the dependency inversion principle with a mechanism called constructor-based dependency injection is applied.

Listing 7.3 contains a modified version of the `PeriphX` structure that was initially introduced in the Single-pass ToD from Listing 7.1. Construct-based dependency injection is used for the provisioning of memory infrastructure: one concrete implementation of the software interface `IMemoryGenerator` is passed to the `__init__` in Line 3.

The first benefit of this pattern is that the user of the generator `PeriphX` can decide which memory generator to provide to `__init__` (as long as it implements the same functionality

```
1 class PeriphX(Structure):
2     def __init__(self, parent: Structure,
3                  testableMemGen: IMemoryGenerator):
4         super().__init__(Ports=[ ... ], parent=parent)
5         ...
6         memX1 = testableMemGen.getMem(self)
7         memX2 = testableMemGen.getMem(self)
8         ...
```

Listing 7.3: Application of Dependency Injection for Infrastructure Generators

and software interface `IMemoryGenerator`). Depending on the concrete implementation, the calls to `getMem` in lines 6 and 7 can behave significantly differently: they can instantiate an actual sub-structure or only return a dummy object to access the read and write ports of the memory which are connected to a corresponding port of `PeriphX`. A consequence of this is that the peripheral generator `PeriphX` is compatible with any memory architecture, as long as a compatible generator is provided.

The second benefit is the dependencies on sub-generators become immediately visible. When a parent generator is used, the constructor arguments define what sub-generators shall be used. This makes multiple, potentially opaque hierarchies of dependencies clearly visible. This is particularly useful in combination with multi-pass generation: the dependency on the multi-pass generator for one of the sub-modules becomes immediately visible to the user of the generator. The user sees that the sub-structure has additional infrastructure needs and is aware that the created structures are not finalized and will be modified in the second and further generation passes.

7.3.3. Management of Multiple Passes

It is not possible to first generate all components which require infrastructure resources and then generate the correct infrastructure resources by inspecting the generated components. The reason for this becomes apparent when looking at the SRAM example again: Assuming there is a generator for memory test controllers that supports two algorithms for memory test X and Y. Both algorithms have very different memory test interfaces (the interface from SRAM wrapper to memory test controller for algorithm X is a lot smaller than the interface for algorithm Y and therefore easier to route in a physical implementation). On the other hand, algorithm X is limited in its coverage and may not meet all application needs, whereas Y can provide extended protection for safety or high-reliability applications. When the overall SoC is generated, the different IP generators have to run in a certain sequence and fully complete the generation of the structures they are responsible for. It is therefore desirable to use memory test algorithm X if no individual IP component requires memory test algorithm Y. For any individual generator X, there is no way of knowing whether one of the other IP generators will require the extended memory test and whether it should construct the SRAM wrapper for algorithms X and Y.

For this purpose, the software interfaces that every multi-pass generator needs to implement are first defined.

```
class IMultipassGenerator(ABC):
    @abstractmethod
    def modifyMoD(self) -> bool:
        """
        Called to run the next generator pass
        :returns: True if changes to the MoD have
                  been made, False otherwise
        """
        pass
```

Listing 7.4: Abstract Base Class of a ToD Multi-pass Generator

Listing 7.4 shows that the software interface `IMultipassGenerator` defines only one single method `modifyMoD`. This method is called by the multi-pass generation library to execute the next pass of MoD construction. The generator that implements this method uses this call to modify the Model-of-Design, adding the infrastructure components it is responsible for. The most important aspect of this method is its return value. This method must return `True` if it has made any changes to the MoD and `False` otherwise. This return value is important as it is used by the multi-pass generation library to handle cyclical dependencies between the generators: it is used to determine when it can stop invoking the generators because the MoD is fully constructed.

7.3.4. Resolution of Circular Dependencies

As mentioned in the previous section, there can be circular dependencies between different infrastructure resources: The memory test controllers have bus interfaces and need to know the characteristics of the bus infrastructure to be constructed, the bus infrastructure in turn needs to be able to raise interrupts and the interrupt controllers may need the memory to allow configurable interrupt priority and buffering of incoming interrupts. The dependencies between the individual generators are circular. There is no linear order in which structures can be generated so that every structure has all its required information available. Based on the defined software interface, multi-pass generators for different infrastructure concerns can be implemented. There can be for example a multi-pass generator for memory test, interrupt handling, bus infrastructure, or handling of aspects of functional safety and security such as management of alarms in a design.

Listing 7.4 shows the API that needs to be used to ensure these generators are executed and the circular dependencies are resolved: the library provides a `MultipassToDRunner` class that is responsible for invoking the multi-pass generators. This class has two key methods: the `registerMultipassGen` method used to register all existing generators (a prerequisite so that the class can later invoke them) and the `runAllPasses` method which eventually runs the generators.

```
1  class MultipassToDRunner:
2      ...
3
4      def registerMultipassGen(self,
5              g: IMultipassGenerator) -> None:
6          """
7          Has to be called to register every
8          IMultipassGenerator in the ToD
9          """
10         ...
11
12     def runAllPasses(self):
13         """
14         This method has to be called after the first
15         pass of ToD execution is complete to run all
16         subsequent passes on the MoD
17         """
18         ...
```

Listing 7.5: Excerpt from Multi-pass ToD Runner Library

The runner will then call the registered multi-pass generators in round-robin style using the `modifyMoD` implemented as part of the `IMultipassGenerator` interface served by each generator. The implementation of `modifyMoD` can then modify the MoD. The return value of this method is used to understand whether the generator pass has modified the MoD. Every modification made by a generator may trigger the need to rerun another generator. The runner will therefore loop over all the multi-pass generators until no further change happens to the MoD by none of the generators.

7.3.5. Example of a Multi-Pass ToD

Listing 7.6 contains a Template-of-Design with support for multi-pass generation of the Model-of-Design. For this example, a `TestableMemoryGenerator` class has been developed which inherits from the abstract base class `IMultipassGenerator` from Listing 7.5. The first major difference can be found in the `main` function, where the `TestableMemoryGenerator` is instantiated. After this, the ToD is executed in a similar manner in the single-pass example. The Major difference here is that the invocation in line 36 will only create the functional aspects and connectivity (1a) and (1b) without the infrastructure components (2a) and (2b). This intermediate state is what is called the first pass of the generator. When this constructor call returns, the MoD is therefore incomplete. Instead of providing a complete MoD, the template is signaling to the multi-pass generator `TestableMemoryGenerator` that there are still open infrastructure needs: the calls to `getMem` in lines 6, 7 and 17 will hand control over to the multi-pass generator responsible for memories. This can also be seen in the UML sequence diagram for the multi-pass generator as provided in Figure 7.3. The multi-pass generator for

memories will create and return an incomplete `Mem` instance and, most importantly, register the need for memory test with the memory generator to be later added. In a realistic example, significantly more details on the properties and requirements for the memory test would be provided so that the memory test generator can later determine the ideal test architecture.

The sequence diagram then sketches the creation of the multi-pass ToD runner which is invoking all generators after the first pass.

In the example, the UML sequence diagram sketches the actions the `modifyMoD` call of the `TestableMemoryGenerator` performs. As all memories have been created when the multipass runner has been created, the generator can then analyze how many memories are created, what their data widths are, and where they are located. Moreover, the ToD will have provided detailed information on the test requirements of the individual memories. Based on this information, it can make the architectural decisions needed to instantiate the memory test: how many test units are needed, on which hierarchies should they be placed, and which test protocols and test design interfaces are needed between the memories and the memory test units. Based on this analysis, the design can be completed, meaning test units can be instantiated, the memory wrappers can be extended with logic and connectivity and the toplevel connections can be created.

7.4. Application and Results

The suggested design pattern was trialed with different infrastructure generators for three categories of infrastructure:

- Different flavors of a memory generator were implemented. A simplified version of these generators has been used to provide the examples in this thesis. The implementation of this generator shows that it is possible to conditionally provide either centralized memory test on design toplevel or distributed memory tests within the IPs and within the subsystems in which the IPs are used. This example further demonstrates that it is possible to switch between memory on top and distributed memory architectures without modifying the peripheral generators.

- The pattern has been applied in a Master's thesis for the generation of SoC Bus Fabric [89]. This work shows that developers of IP generators can eliminate any need to address bus-specific topics such as the instantiation of bus interface modules or the creation of any bus interfaces. A key finding of this work is that it is important to define clear interfaces between the peripherals requiring the infrastructure (HW/SW interfaces for example for configuration registers or data streams) and the bus generators.

- A simple interrupt controller has been implemented to show how this distributed generator approach can both provide wires to IP generators which can be used to signal interrupts and also perform toplevel connectivity and interrupt handling for these wires.

```
class PeriphX(Structure):
    def __init__(self, testableMemGen, parent):
        super().__init__(Ports=[ ... ], parent=parent)
                        # create ports for (1b) only
        ...
        memX1 = testableMemGen.getMem(self)
        memX2 = testableMemGen.getMem(self)
        ...
        # end of constructor, created substructures,
        # logic, and connectivity for (1a) only

class PeriphY(Structure):
    def __init__(self, testableMemGen, parent):
        super().__init__(Ports=[ ... ], parent=parent)
                        # create ports for (1b) only
        ...
        memY1 = testableMemGen.getMem(self)
        ...
        # end of constructor, created substructures,
        # logic, and connectivity for (1a) only

class Toplevel(Structure):
    def __init__(self, testableMemGen, parent):
        super().__init__(Ports=[ ... ], parent=parent)
                        # create ports for (1b) only
        ...
        pX = PeriphX(testableMemGen, self)
        pY = PeriphY(testableMemGen, self)
        ...
        # end of constructor, created substructures,
        # logic, and connectivity for (1a) only

def main(Api, MoD):
    testableMemGen = TestableMemoryGenerator(MoD)

    Toplevel(testableMemGen, parent=MoD)

    todRunner = MultipassToDRunner()
    todRunner.registerMultipassGen(testableMemGen)
    todRunner.runAllPasses() # ports for (2a) and (2b),
    # potentially centralized memory test units
```

Listing 7.6: Sample Template-of-Design with Multi-pass Generation

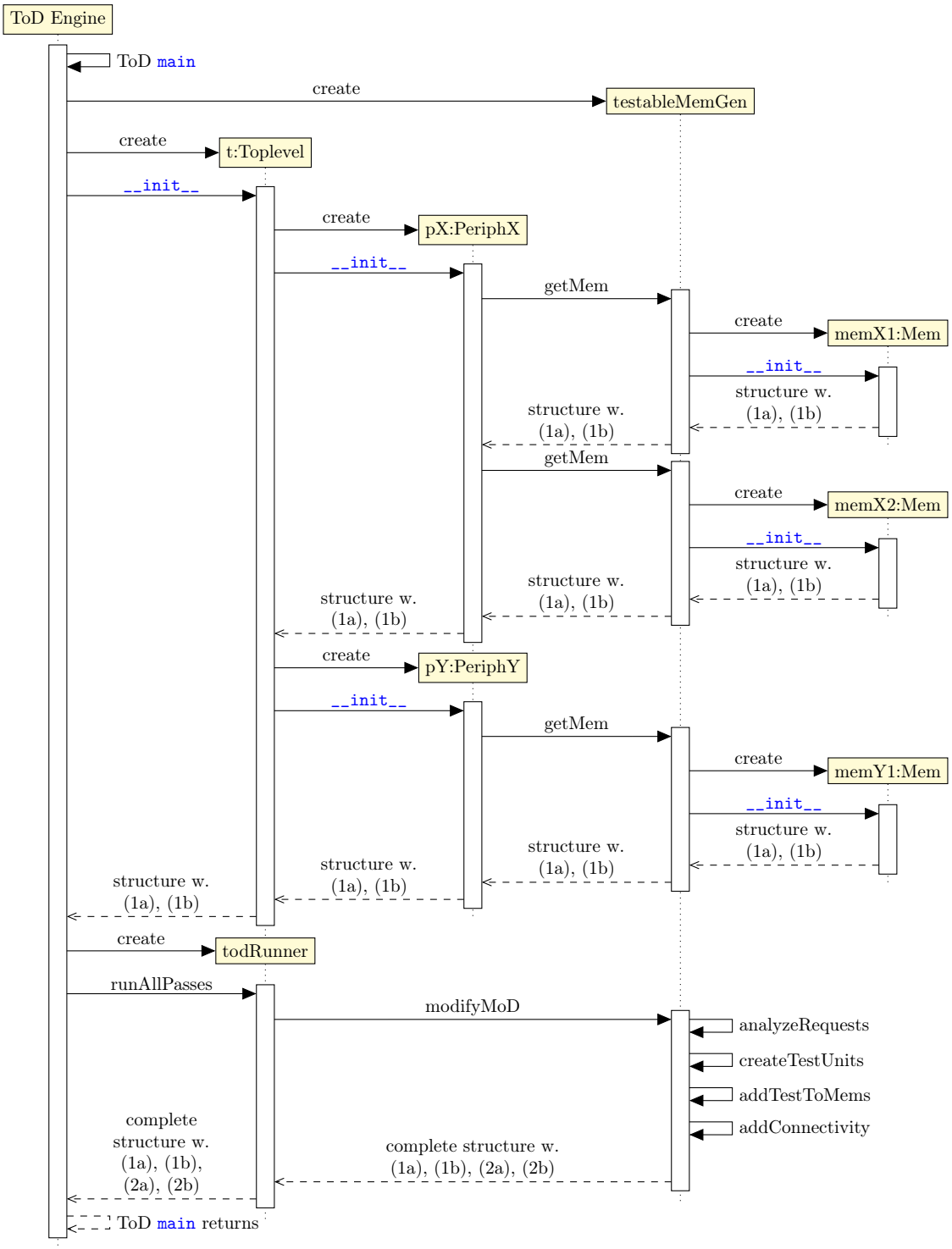

Figure 7.3.: UML Sequence Diagram of Template-of-Design in Listing 7.6 with Multi-pass Generation

It is possible to confirm the benefits of the suggested multi-pass flow with the examples of bus fabric, memory test, and interrupt routing. The combination of different infrastructure generators also showed that the suggested patterns can handle circular dependencies: the centralized memory test units were connected to the bus infrastructure and the information about the required connection was only available after the second pass of the memory generator.

7.4.1. Key Achievements

Based on the pattern presented in this chapter, it is possible to develop IP generators entirely independent of SoC integration aspects. The provisioning of SoC integration inside the generated IPs (e.g. optimal bus interface sub-components) is transparently delegated to specialized generators. Changes to the centralized generators can be performed that result in changes to the generated hardware can and the protocols utilized in it can happen without modifications to the IP generators intended for reuse. The centralized handling of the multiple passes of these generators has been demonstrated to be a reliable method to ensure that all infrastructure elements are provided in the design.

It was found that it is straightforward for infrastructure generators to automatically collect information on the needs of the individual IP generators generated. Using this information, the infrastructure generators can automatically and transparently generate suitable SoC infrastructure (e.g. bus fabric with topology adjusted to the individual generators) and provide the toplevel connectivity for their infrastructure. This simplification becomes particularly significant in larger SoCs. Here, many infrastructure needs typically propagate to the toplevel and require complex centralized connectivity and provisioning of modules that provide the infrastructure. The approach presented in this chapter allows to apply a clean divide-and-conquer approach where the different centralized needs are handled by reusable and modular generators.

7.4.2. Challenges

Based on the application of the patterns presented in this chapter and based on the experience with the application of model-based hardware generation in an industrial context, two challenges can be seen that need to be addressed to successfully roll-out such a methodology.

7.4.2.1. Adaptation to Software Thinking and Software Methodology

Throughout the research on model-based generation methods for digital design, it was found that the transition from traditional hardware description languages to hardware generation approaches comes with a learning curve that is particularly difficult for hardware designers to overcome. It is a major focus of the MDA framework to be simple and intuitive to use. This is reflected in the transparent and easy-to-learn metamodeling framework that is applied to

implement it. The choice of the programming language Python is equally important for this: it is a language with a particularly flat learning curve that is commonly utilized outside of the scope of professional software engineering and taught as part of university curricula in the field of science, technology, and engineering. This has helped the adaptation of the methodology, particularly in an environment of hardware designers who are already familiar with Python scripting.

While the introduction of the patterns suggested in this chapter shows real benefits and significant potential above the productivity benefits already identified with the current generator development, it also makes the introduction of the methodology to new engineers more challenging. For successful adoption of the methodology, it is considered essential to provide good training and ensure all utilized design patterns are the *simplest-in-class* rather than *best-in-class* approaches. The methodology therefore explicitly avoids making use of sophisticated software libraries such as dependency injection frameworks.

7.4.2.2. Definition of Reusable and Stable Interfaces for Infrastructure Requests

The definition of clear, clean, and consistent software interfaces that allow a modular overall architecture and their application to generator development is extremely powerful. They are the foundation required for fully reusable IP generators which do not require any changes in the generator source code when used in different integration contexts. It is however important to state that when this pattern is consistently applied, the IP generators depend on the software interface instead of depending on the component implementing it.

Any changes to these software interfaces which are not backward-compatible will require changing the generators that make use of them. The definition of these software interfaces, therefore, has to be done with great caution and in consideration of all potential use cases. Poor quality of these interface definitions will inevitably lead to technical debt that is hard to resolve.

8. Summary and Key Contributions

This thesis adapts, transfers, and extends the vision of Model Driven Architecture for use in the field of digital hardware design. It identifies key Metamodels and Model-to-Model transformations and implements an end-to-end generation flow for digital designs. The framework developed in the context of this thesis puts particular focus on Metamodel-based automation to reduce the manual effort required to develop generators.

The new framework enables a shift from the development of static instances of digital hardware in hardware description languages such as SystemVerilog and VHDL to the development of highly configurable generators in a general-purpose software programming language. This thesis demonstrates how the capabilities of this general-purpose programming language and the MDA-inspired modeling environment can be applied for generator reuse and to solve complex System-on-Chip (SoC) infrastructure problems.

In the timespan between the start of this thesis work and its completion, the research and implementation provided by it have laid the foundation for what is now the MetaX initiative at Infineon Technologies AG. As part of this initiative, the work of this thesis is applied outside of academia and in an industrial context. MetaX is a key element of the strategic path Infineon takes towards automation and code generation and developing into an integral component in Infineon's design flow. The following paragraphs contain the key contributions of this thesis to the MetaX initiative.

8.1. Key Contributions

Formalization of MDA in the HW domain This thesis provides a firm formal framework for the application of Model Driven Architecture to the hardware domain. It uses the vision of Model Driven Architecture that was developed by the Object Management Group (OMG) for the software domain and defines a corresponding approach for hardware development. As part of this approach, a schema of layers with clear terminologies is developed that precisely captures the intention of the MDA layers of the hardware domain. It defines the level of abstraction of these layers and their semantics.

This foundational work is applied to the automated generation of digital designs in this thesis. The formal framework that has been developed has also been the basis for a body of research conducted by the Infineon Metamodeling group. In this context, the approach has been successfully applied outside of digital design: it has been applied to formal verification automation and the development of firmware and embedded software implementation automation.

Definition of Key Metamodels This thesis defines key Metamodels for Digital Design on RTL level. The Metamodel introduced for the so-called "Model-of-Design" layer is called *MetaRTL*. The structural approach to representing a design on register transfer level differs significantly from related work introduced in Chapter 3 and demonstrates measurable benefits over it. In addition, this thesis defines the level of abstraction and Metamodel on the so-called "Model-of-View" layer. Moreover, this thesis provides *Orthogonal Metamodels* that are powerful enablers for use on different levels of abstraction. They are applied and their success is demonstrated in the context of this thesis. Moreover, they are applied to related work that is based on the foundation provided by this thesis – acting as a common enabler for Model Driven Architecture Methodology in the hardware domain.

Identification of a Novel Approach to View Generation This thesis further introduces a significant innovation in the field of view generation. It defines and implements the so-called *View Language Description (VLD)* format as a way to automatically generate the definition of the "Model-of-View" layer from an EBNF-like description of the target view's grammar. With the VLD format as input, the process of view generation from "Model-of-View" layer has been 100% automated. The approach taken here is novel and significantly differs from existing approaches to code generation. The publication that introduces VLD has been awarded the "best poster award" of the conference.

This approach is applied in this thesis as it eases the transfer of structural design information onto the event-driven simulation semantics of the target language. Aside from the practical benefits, this thesis also illustrates how the VLD-based approach automatically leads to clean, consistently formatted, and grammatically correct target views, simplifying the utilization of the generated views in design flows and eliminating an entire class of tool bugs: the accidental generation of target views with incorrect grammar. The beneficial nature of the novel approach is also underlined by the fact that research based on this thesis has successfully applied this approach to other domains [85, 97].

Implementation of a Model Driven Architecture Framework The formalization and approaches presented so far are entirely independent of a certain target language and only depend on the generic properties of common Metamodeling frameworks. This thesis provides an implementation of the MDA framework based on Infineon's proprietary Metamodeling Framework. It develops key components to extend the existing framework to enable its use in a Model Driven Architecture context.

Infrastructure for Powerful Model-to-Model Transformations This thesis identifies the efficiency of Model-to-Model transformations as a key area of optimization for Model Driven Architecture approaches. It develops and promotes an imperative approach to Model-to-Model transformation. This approach significantly differs from other approaches such as the commonly applied declarative Extensible Stylesheet Language Transformation (XSLT) or Epsilon Transformation Language (ETL) approaches. The benefits of this approach become particu-

larly apparent when comparing the efficiency of the Model-to-Model transformations to related work. This thesis demonstrates that the capabilities exceed the state of what can be provided by today's hardware description languages and are on par with what the field or research of Hardware Generation Languages (HGLs) can provide. The approach is superior to existing HGL approaches in three aspects.

First, it is a *Model-to-Model transformation approach*. The HGL approach can be at best interpreted as a *Model-Generation approach*. Having input models as an elementary part of the approach is what eventually enables the full end-to-end automation essential to the Model Driven Architecture approach. Such automation is not a primary aspect of today's HGL research.

Second, the Python-based approach provides access to the near-infinite toolbox of Python-based libraries and tools right during the Model-to-Model transformation. This enables nothing less than the conflation of hardware design and design space exploration.

Third, the approach is a lot easier to learn and understand than the existing approaches in the field of HGLs. It relies on a framework centered around Python, enabling a fast learning curve for engineers outside the field of software development. The Metamodels that are defined for all layers of the Model Driven Architecture framework also have a significant contribution to the ease of use of the framework: The visual representation of the Metamodel and its formalisms allow engineers to get started with constructing Model-to-Model transformations right away – learning any Domain-Specific Languages is not required.

Application of Software Engineering Methodology to Digital Design The Model Driven Architecture framework developed in this thesis opens the world of digital hardware design to the patterns and principles that are established in Software Engineering. All development of hardware and hardware generators based on the framework is conducted as the development of Model-to-Model transformations. This development happens on a language and platform designed for software development. Most noteworthy, the SOLID principles developed for understandable, flexible, and extendable software architecture are applied to the world of IC design. This transfer provides capabilities to the domain of IC design that has so far not been available, opening up the path for an entirely new approach to design reuse.

The outcome of this thesis has been translated into practical applications in industrial designs at Infineon Technologies AG, spanning from FPGAs to full-custom ICs. These real-world applications show that the work of this thesis has surpassed the confines of academia and yielded an industrial-strength solution.

8.2. Publications

Large parts of this thesis, including its key findings and the novel solutions proposed and developed in the context of this thesis, have already been published in several scientific conferences and journals [1, 2, 3, 4, 5, 6]. Any reuse of the author's own work from these previous publications in this thesis is not marked with citations that attribute it to the author's own work. Instead, the original source is cited wherever applicable.

[1] Johannes Schreiner, Rainer Findenig, and Wolfgang Ecker. "Design centric modeling of digital hardware". In: *2016 IEEE International High Level Design Validation and Test Workshop (HLDVT)*. 2016, pp. 46–52. DOI: 10.1109/HLDVT.2016.7748254.

[2] Keerthikumara Devarajegowda, Johannes Schreiner, and Wolfgang Ecker. "Python Based Framework for HDSLs with an Underlying Formal Semantics". In: *Proceedings of the 36th International Conference on Computer-Aided Design*. ICCAD '17. Irvine, California: IEEE Press, Nov. 2017, pp. 1019–1025. DOI: 10.1109/ICCAD.2017.8203893.

[3] Wolfgang Ecker and Johannes Schreiner. "Metamodeling and Code Generation in the Hardware/Software Interface Domain". In: *Handbook of Hardware/Software Codesign*. Ed. by Soonhoi Ha and Jürgen Teich. Dordrecht: Springer Netherlands, 2017, pp. 1051–1091. ISBN: 978-94-017-7267-9. DOI: 10.1007/978-94-017-7267-9_32. URL: https://doi.org/10.1007/978-94-017-7267-9_32.

[4] Johannes Schreiner and Wolfgang Ecker. "Digital Hardware Design Based on Metamodels and Model Transformations". In: *VLSI-SoC: System-on-Chip in the Nanoscale Era – Design, Verification and Reliability*. Ed. by Thomas Hollstein et al. Cham: Springer International Publishing, Sept. 2017, pp. 83–107. ISBN: 978-3-319-67104-8.

[5] Johannes Schreiner, Felix Willgerodt, and Wolfgang Ecker. "A New Approach for Generating View Generators". In: *Proceedings of DVCON US 2017*. DVCON US 2017. IEEE Press, Feb. 2017.

[6] Johannes Schreiner et al. "Generator IP-reuse and Automated Infrastructure Generation for Model-based Full Chip Generation". In: *MBMV 2023; 26th Workshop*. accepted and presented, currently in publication, 2023, pp. 1–8.

Bibliography

[7] Laurence W. Nagel and D.O. Pederson. *SPICE (Simulation Program with Integrated Circuit Emphasis)*. Tech. rep. UCB/ERL M382. EECS Department, University of California, Berkeley, Apr. 1973. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/1973/22871.html.

[8] Hilary J Kahn and Richard F Goldman. "The electronic design interchange format edif: present and future". In: *[1992] Proceedings 29th ACM/IEEE Design Automation Conference*. IEEE. 1992, pp. 666–671.

[9] Peggy Salz Trautman. *A Computer Pioneer Rediscovered, 50 Years On*. 1994. URL: https://web.archive.org/web/20161104051054/http://www.nytimes.com/1994/04/20/news/20iht-zuse.html (visited on 11/04/2016).

[10] "IEEE Standard Hardware Description Language Based on the Verilog(R) Hardware Description Language". In: *IEEE Std 1364-1995* (1996), pp. 1–688. DOI: 10.1109/IEEESTD.1996.81542.

[11] Daniel D. Gajski. *Principles of Digital Design*. Pearson, 1997.

[12] Bertrand Meyer. *Object-Oriented Software Construction*. 2nd ed. Upper Saddle River, NJ: Prentice Hall, 1997. ISBN: 978-0-13-629155-8.

[13] R.R. Schaller. "Moore's law: past, present and future". In: *IEEE Spectrum* 34.6 (1997), pp. 52–59. DOI: 10.1109/6.591665.

[14] P. Bellows and B. Hutchings. "JHDL-an HDL for reconfigurable systems". In: *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines (Cat. No. 98TB-100251)*. 1998, pp. 175–184. DOI: 10.1109/FPGA.1998.707895.

[15] Don Mills and Clifford E Cummings. "RTL coding styles that yield simulation and synthesis mismatches". In: *SNUG (Synopsys Users Group) 1999 Proceedings*. 1999.

[16] James Jennings and Eric Beuscher. "Verischemelog: Verilog Embedded in Scheme". In: *SIGPLAN Not.* 35.1 (Dec. 2000), pp. 123–134. ISSN: 0362-1340. DOI: 10.1145/331963.331978. URL: https://doi.org/10.1145/331963.331978.

[17] Yanbing Li and Miriam E. Leeser. "HML, a novel hardware description language and its translation to VHDL". In: *IEEE Trans. Very Large Scale Integr. Syst.* 8 (2000), pp. 1–8.

[18] J. Bezivin and O. Gerbe. "Towards a precise definition of the OMG/MDA framework". In: *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. 2001, pp. 273–280. DOI: 10.1109/ASE.2001.989813.

[19] Jack St Clair Kilby. "Turning potential into realities: The invention of the integrated circuit (Nobel lecture)". In: *ChemPhysChem* 2.8-9 (2001), pp. 482–489.

[20] Stephen J Mellor et al. "Model-driven architecture". In: *Advances in Object-Oriented Information Systems*. Springer, 2002, pp. 290–297.

[21] Colin Atkinson and Thomas Kühne. "Model-driven development: a metamodeling foundation. IEEE Software 20(5), 36-41". In: *Software, IEEE* 20 (Oct. 2003), pp. 36–41. DOI: 10.1109/MS.2003.1231149.

[22] Krzysztof Czarnecki and Simon Helsen. "Classification of Model Transformation Approaches". In: Jan. 2003.

[23] Anneke G Kleppe et al. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003.

[24] Robert C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003. URL: http://dl.acm.org/citation.cfm?id=515230.

[25] MDA OMG. "Guide Version 2.0. rev. 1". In: *Object Management Group* 62 (2003), p. 34. URL: http://www.omg.org/cgi-bin/doc?ormsc/14-06-01.

[26] S. Sendall and W. Kozaczynski. "Model transformation: the heart and soul of model-driven software development". In: *IEEE Software* 20.5 (2003), pp. 42–45. DOI: 10.1109/MS.2003.1231150.

[27] Alan W Brown. "Model driven architecture: Principles and practice". In: *Software and systems modeling* 3.4 (2004), pp. 314–327.

[28] J. Martin and J. Gutenberg. "Automated source code transformations on fourth generation languages". In: *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.* 2004, pp. 214–220. DOI: 10.1109/CSMR.2004.1281422.

[29] Erik Meijer and Peter Drayton. "Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages". In: Jan. 2004.

[30] R. Nikhil. "Bluespec System Verilog: efficient, correct RTL from high level specifications". In: *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.* 2004, pp. 69–70. DOI: 10.1109/MEMCOD.2004.1459818.

[31] Martin Odersky et al. "An overview of the Scala programming language". In: (2004).

[32] Frank P Coyle and Mitchell A Thornton. "From UML to HDL: a model driven architectural approach to hardware-software co-design". In: *Information systems: new generations conference (ISNG)*. Vol. 1. 2005, pp. 88–93.

[33] Martin Heimann, Wolfgang Käfer, and Oliver Kraus. "MDA Success Story – ePEP successful with Model Driven Architecture". In: (2005).

[34] Martin Kempa and Zoltán Adám Mann. "Model driven architecture". In: *Informatik-Spektrum* 28.4 (2005), pp. 298–302.

[35] Marjan Mernik, Jan Heering, and Anthony M Sloane. "When and how to develop domain-specific languages". In: *ACM computing surveys (CSUR)* 37.4 (2005), pp. 316–344.

[36] V. Berman. "Standards: The P1685 IP-XACT IP Metadata Standard". In: *IEEE Design and Test of Computers* 23.4 (2006), pp. 316–317. DOI: 10.1109/MDT.2006.104.

[37] Michi Henning. "The Rise and Fall of CORBA: There's a Lot We Can Learn from CORBA's Mistakes." In: *Queue* 4.5 (2006), pp. 28–34. ISSN: 1542-7730. DOI: 10.1145/1142031.1142044. URL: https://doi.org/10.1145/1142031.1142044.

[38] Frédéric Jouault et al. "ATL: a QVT-like transformation language". In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. 2006, pp. 719–720.

[39] K. Keutzer et al. "System-level Design: Orthogonalization of Concerns and Platform-based Design". In: *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 19.12 (Nov. 2006), pp. 1523–1543. ISSN: 0278-0070. DOI: 10.1109/43.898830.

[40] Tom Mens and Pieter Van Gorp. "A Taxonomy of Model Transformation". In: *Electronic Notes in Theoretical Computer Science* 152 (2006). Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005), pp. 125–142. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2005.10.021. URL: https://www.sciencedirect.com/science/article/pii/S1571066106001435.

[41] Roland Petrasch and Oliver Meimberg. "Model Driven Architecture". In: *Heidelberg: dpunkt. verlag* (2006).

[42] F. Truyen. "The fast Guide to Model Driven Architecture". In: (2006). URL: http://www.omg.org/mda/mda_files/Cephas_MDA_Fast_Guide.pdf (visited on 02/08/2016).

[43] Franck Chauvel and Franck Fleurey. *Kermeta Language Overview*. 2007.

[44] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*. Springer. 2008, pp. 337–340.

[45] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. Polack. "The Epsilon Transformation Language". In: *Proceedings of the 1st International Conference on Theory and Practice of Model Transformations*. ICMT '08. Zurich, Switzerland: Springer-Verlag, 2008, pp. 46–60. ISBN: 9783540699262. DOI: 10.1007/978-3-540-69927-9_4. URL: https://doi.org/10.1007/978-3-540-69927-9_4.

[46] Wido Kruijtzer et al. "Industrial IP integration flows based on IP-XACT standards". In: *Design, Automation and Test in Europe, 2008. DATE'08*. IEEE. 2008, pp. 32–37.

[47] Wido Kruijtzer et al. "Industrial IP integration flows based on IP-XACT™ standards". In: *Proceedings of the conference on Design, automation and test in Europe*. 2008, pp. 32–37.

[48] Fabrice Marguerie, Steve Eichert, and Jim Wooley. *LINQ in Action*. Manning Publications Co., 2008.

[49] D. Steinberg et al., eds. *EMF Modeling Framework*. Addison Wesley, 2008.

[50] Doug Tidwell. *XSLT: mastering XML transformations*. " O'Reilly Media, Inc.", 2008.

[51] J. Mangler, E. Schikuta, and C. Witzany. "Quo vadis interface definition languages? Towards a interface definition language for RESTful services". In: *2009 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*. 2009, pp. 1–4. DOI: 10.1109/SOCA.2009.5410459.

[52] Grant Martin and Gary Smith. "High-Level Synthesis: Past, Present, and Future". In: *IEEE Des. Test* 26.4 (2009), pp. 18–25. ISSN: 0740-7475. DOI: 10.1109/MDT.2009.83. URL: https://doi.org/10.1109/MDT.2009.83.

[53] Alberto Sangiovanni-Vincentelli et al. "Metamodeling: An Emerging Representation Paradigm for System-Level Design". In: *IEEE Design & Test of Computers* 26.3 (2009), pp. 54–69. DOI: 10.1109/MDT.2009.62.

[54] "ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary". In: *ISO/IEC/IEEE 24765:2010(E)* (2010), pp. 1–418. DOI: 10.1109/IEEESTD.2010.5733835.

[55] Rishiyur S Nikhil and Kathy R Czeck. "BSV by Example". In: *CreateSpace, Dec* (2010).

[56] O. Shacham et al. "Rethinking Digital Design: Why Design Must Change". In: *IEEE Micro* 30.6 (2010), pp. 9–24.

[57] Wilson Snyder. "The Verilog Preprocessor: Force for Good and Evil". In: 2010.

[58] J. Bachrach et al. “Chisel: constructing hardware in a Scala embedded language”. In: *The 49th Annual Design Automation Conference 2012, DAC ’12, San Francisco, CA, USA, June 3-7, 2012*. 2012, pp. 1216–1225.

[59] Sebastian Kleinschmager et al. “Do static type systems improve the maintainability of software systems? An empirical study”. In: *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. 2012, pp. 153–162. DOI: 10.1109/ICPC.2012.6240483.

[60] Moomen Chaari. “Implementation of a Template Rendering Mechanism based on Model-driven C++ Code Generation”. Technische Universität München, 2013.

[61] R. Collet and D. Pyle. *McKinsey on Semiconductors: What happens when chip-design complexity outpaces development productivity*. 2013.

[62] Stefan Hanenberg et al. “An empirical study on the impact of static typing on software maintainability”. In: *Empirical Software Engineering* 19 (Oct. 2013). DOI: 10.1007/s10664-013-9289-1.

[63] Wolfgang Ecker et al. “The metamodeling approach to system level synthesis.” In: *DATE*. Ed. by Gerhard Fettweis and Wolfgang Nebel. European Design and Automation Association, 2014, pp. 1–2. ISBN: 978-3-9815370-2-4.

[64] Scott Meyers. *Effective modern C++: 42 specific ways to improve your use of C++ 11 and C++ 14*. " O’Reilly Media, Inc.", 2014.

[65] Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft. *Java 8 in action*. Manning publications, 2014.

[66] T. Gottardi and Rosana Braga. “Model driven development success cases for domain-specific and general purpose approaches a systematic mapping”. In: *CIBSE 2015 - XVIII Ibero-American Conference on Software Engineering* (Jan. 2015), pp. 432–445.

[67] B. Nikolic. “Simpler, more efficient design”. In: *ESSCIRC Conference 2015 - 41st European Solid-State Circuits Conference, Graz, Austria, September 14-18, 2015*. 2015, pp. 20–25.

[68] Sebastian Schelle. “A Structural Intermediate for RTL Code Generation”. MA thesis. Technische Universität München, 2015.

[69] "OMG". *MDA - The Architecture of Choice for a Changing World*. 2016. URL: http://www.omg.org/mda/ (visited on 02/08/2016).

[70] Patrick S. Li, Adam M. Izraelevitz, and Jonathan Bachrach. *Specification for the FIRRTL Language*. Tech. rep. 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.

[71] Pierre-André Mudry. *SpinalHDL Documentation*. 2016. URL: https://github.com/pmudry/SpinalDoc/blame/6489a25/manual/userGuide.md (visited on 12/30/2022).

[1] Johannes Schreiner, Rainer Findenig, and Wolfgang Ecker. “Design centric modeling of digital hardware”. In: *2016 IEEE International High Level Design Validation and Test Workshop (HLDVT)*. 2016, pp. 46–52. DOI: 10.1109/HLDVT.2016.7748254.

[2] Keerthikumara Devarajegowda, Johannes Schreiner, and Wolfgang Ecker. “Python Based Framework for HDSLs with an Underlying Formal Semantics”. In: *Proceedings of the 36th International Conference on Computer-Aided Design*. ICCAD ’17. Irvine, California: IEEE Press, Nov. 2017, pp. 1019–1025. DOI: 10.1109/ICCAD.2017.8203893.

[3] Wolfgang Ecker and Johannes Schreiner. “Metamodeling and Code Generation in the Hardware/Software Interface Domain”. In: *Handbook of Hardware/Software Codesign*.

Ed. by Soonhoi Ha and Jürgen Teich. Dordrecht: Springer Netherlands, 2017, pp. 1051–1091. ISBN: 978-94-017-7267-9. DOI: 10.1007/978-94-017-7267-9_32. URL: https://doi.org/10.1007/978-94-017-7267-9_32.

[72] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Robert C. Martin Series. Boston, MA: Prentice Hall, 2017. ISBN: 978-0-13-449416-6.

[4] Johannes Schreiner and Wolfgang Ecker. "Digital Hardware Design Based on Metamodels and Model Transformations". In: *VLSI-SoC: System-on-Chip in the Nanoscale Era – Design, Verification and Reliability*. Ed. by Thomas Hollstein et al. Cham: Springer International Publishing, Sept. 2017, pp. 83–107. ISBN: 978-3-319-67104-8.

[5] Johannes Schreiner, Felix Willgerodt, and Wolfgang Ecker. "A New Approach for Generating View Generators". In: *Proceedings of DVCON US 2017*. DVCON US 2017. IEEE Press, Feb. 2017.

[73] Simon Schwichtenberg, Christian Gerth, and Gregor Engels. "From open API to semantic specifications and code adapters". In: *2017 IEEE International Conference on Web Services (ICWS)*. IEEE. 2017, pp. 484–491.

[74] "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language". In: *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)* (2018), pp. 1–1315. DOI: 10.1109/IEEESTD.2018.8299595.

[75] Tutu Ajayi et al. "OpenROAD: Toward a Self-Driving, Open-Source Digital Layout Implementation Tool Chain". In: 2019.

[76] "IEEE Standard for VHDL Language Reference Manual". In: *IEEE Std 1076-2019* (2019), pp. 1–673. DOI: 10.1109/IEEESTD.2019.8938196.

[77] A. Ronzhin and V. Shishlakov. *Proceedings of 14th International Conference on Electromechanics and Robotics "Zavalishin's Readings": ER(ZR) 2019, Kursk, Russia, 17 - 20 April 2019*. Smart Innovation, Systems and Technologies. Springer Singapore, 2019. ISBN: 9789811392672. URL: https://books.google.de/books?id=HS2sDwAAQBAJ.

[78] Paul Vincent et al. "Magic quadrant for enterprise low-code application platforms". In: *Gartner report* (2019).

[79] Thomas Bourgeat, Clément Pit-Claudel, and Adam Chlipala. "The essence of Bluespec: a core language for rule-based hardware design". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2020, pp. 243–257.

[80] The Eclipse Foundation. *The Epsilon Transformation Language (ETL)*. 2020. URL: https://github.com/eclipse/epsilon-website/blob/d117eba/mkdocs/docs/doc/etl.md (visited on 12/25/2022).

[81] Ahmed Alaa Ghazy and Mohamed Shalan. "OpenLANE: The Open-Source Digital ASIC Implementation Flow". In: 2020.

[82] Mamillapally Raghavender Sharma. "A Short Communication on Computer Programming Languages in Modern Era". In: *International Journal of Computer Science and Mobile Computing* 9 (Sept. 2020), pp. 50–60. DOI: 10.47760/IJCSMC.2020.v09i09.006.

[83] Apurvanand Sahay et al. "Supporting the understanding and comparison of low-code development platforms". In: *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE. 2020, pp. 171–178.

[84] Sandra Casas et al. "Uses and applications of the OpenAPI/Swagger specification: a systematic mapping of the literature". In: *2021 40th International Conference of the Chilean Computer Science Society (SCCC)*. IEEE. 2021, pp. 1–8.

[85] Keerthikumara Devarajegowda. "HW-Acceleration for Edge-AI". PhD thesis. Technische Universität München, 2021.

[86] Diogo Manuel Gonçalves Romão. "Migration from Legacy to Reactive Applications in OutSystems". PhD thesis. Universidade NOVA de Lisboa (Portugal), 2021.

[87] Sourabh Sharma. *Modern API Development with Spring and Spring Boot: Design highly scalable and maintainable APIs with REST, gRPC, GraphQL, and the reactive paradigm*. Packt Publishing Ltd, 2021.

[88] Inc Anaconda. *Package, dependency and environment management for any language, rev. cd0261c4*. 2022. URL: https://docs.conda.io/en/latest/ (visited on 12/27/2022).

[89] Vasundhara Raje Gontia. "Configurable and Extendable SoC Bus System and Interface Generator". MA thesis. Munich University of Applied Sciences, 2022.

[90] Christian Lück et al. "Industrial Experience with Open-Source EDA Tools". In: *2022 ACM/IEEE 4th Workshop on Machine Learning for CAD (MLCAD)*. 2022, pp. 143–143. DOI: 10.1109/MLCAD55463.2022.9900097.

[91] Jeremy Norman. *The 24 AN/FSQ-7 Computers IBM Built for SAGE are Physically the Largest Computers Ever Built*. 2022. URL: https://www.historyofinformation.com/detail.php?id=751 (visited on 12/17/2022).

[92] Gerhard Petrowitsch. *Hyperlinked VHDL-93 BNF Syntax*. 2022. URL: https://tams-www.informatik.uni-hamburg.de/vhdl/tools/grammar/vhdl93-bnf.html (visited on 01/04/2023).

[93] Ofer Shacham. "Chip Multiprocessor Generator: Automatic Generation Of Custom and Heterogeneous Compute Platforms". In: (Dec. 2022).

[94] Nico Vermeir. ".NET Compiler Platform". In: *Introducing .NET 6: Getting Started with Blazor, MAUI, Windows App SDK, Desktop Development, and Containers*. Berkeley, CA: Apress, 2022, pp. 275–295. ISBN: 978-1-4842-7319-7. DOI: 10.1007/978-1-4842-7319-7_10. URL: https://doi.org/10.1007/978-1-4842-7319-7_10.

[95] Juan Cruz Viotti and Mital Kinderkhedia. *A Survey of JSON-compatible Binary Serialization Specifications*. 2022. DOI: 10.48550/ARXIV.2201.02089. URL: https://arxiv.org/abs/2201.02089.

[96] Bill Wagner. *Roslyn Source Generators Overview*. https://github.com/dotnet/docs/blob/08f8365/docs/csharp/roslyn-sdk/source-generators-overview.md. 2022.

[97] Michael Werner. "Automatic Generator Methodology for Safe Embedded Software". 2022.

[98] Michael Bayer. *Mako Template Engine*. 2023. URL: https://www.makotemplates.org (visited on 04/22/2023).

[99] Jimmy Bogard. *AutoMapper Documentation, Revision 4443a59f*. 2023. URL: https://docs.automapper.org/en/latest/ (visited on 01/11/2023).

[100] TIOBE Software BV. *TIOBE Index for April 2023*. 2023. URL: https://www.tiobe.com/tiobe-index/ (visited on 04/16/2023).

[101] Inc Eclipse Foundation. *Epsilon Playground ETL Examples*. 2023. URL: https://www.eclipse.org/epsilon/playground/?etl (visited on 04/22/2023).

[102] PYPL project. *PYPL PopularitY of Programming Language*. 2023. URL: https://pypl.github.io/PYPL.html (visited on 04/16/2023).

[6] Johannes Schreiner et al. “Generator IP-reuse and Automated Infrastructure Generation for Model-based Full Chip Generation”. In: *MBMV 2023; 26th Workshop*. accepted and presented, currently in publication, 2023, pp. 1–8.

[103] Liangora Research Lab. *What is MDA? Why considering BNPM*. URL: `https://research.linagora.com/pages/viewpage.action?pageId=3639295` (visited on 04/25/2016).