Digital Hardware Design based on Metamodels and Model Transformations

Johannes Schreiner and Wolfgang Ecker

Infineon Technologies AG / Technische Universität München
<first name>.<last name>@infineon.com
<first name>.<last name>@tum.de

Abstract. This contribution presents a Model-driven Architecture (MDA) inspired strategy for the automation of digital hardware design starting at specification level and targeting RT-level. This strategy defines a structured approach with is superior to code generation using scripts, print statements or template engines directly targeting ASCII files.
As part of this strategy, we implemented intermediate models named Models-of-Things (MoTs) for formalizing specification data which have a dependency on the design objects specified. We further implemented a Model-of-Design (MoD) for hardware design related modeling, and a Model-of-View (MoV) for target view generation. For the transformations between our intermediate models, we use a template based approach. These templates are executed and generate more concrete models utilizing information from more abstract models. The term model here describes instances of Metamodels in the terminology of MDA and must not be mixed up with simulation and synthesis models such as VHDL RTL models.
The template which guides MoD generation is called Template-of-Design. On one hand, the Template-of-Design (ToD) captures the (micro-)architecture and on the other hand, it retrieves MoT data to automate creation of the design that meets the specification. The Template-of-Design is Python code – as all our framework is implemented in Python – and uses generated APIs to simplify (micro-)architecture construction. In contrast templates for classical template engines, the Template-of-Design generates a model and not an ASCII File.
To generate the final view, a so-called Template-of-View (ToV) is used. It is also encoded as Python code, traverses the Model-of-Design and creates the MoV. This Model-of-View is in many aspects similar to Abstract Syntax Trees utilized in language parsing. It is defined by a Metamodel, which is generated from our so-called View Language Description (VLD). This language is also used to generate an un-parse method, which automates the view generation task from MoV instances. This means that provided our VLD and a Model-of-View instance, we can generate an ASCII-File containing e.g. RTL VHDL code in a completely automated way. Our VLD format also includes formatting pragmas to guide view generation tasks such as indentation and alignment.
Our strategy is supported by type and expression Metamodels that are used across different Models on different modeling layers (i.e. in MoTs or MoDs as well as in the Template-of-Design). This further simplifies implementation of the models as well as of the templates.
First analysis of our approach shows that we can simplify building one generator up to a factor of 10x. This factor increases further when different target languages or target language styles are generated from one MoD.

Keywords: Model-driven Architecture, Hardware Generation, Metamodeling

1 Introduction

In the 2013 McKinsey study on semiconductors [4], Collet et al. analyze how the *design gap* widened by a factor of about 5x in one decade. 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. McKinsey concluded that closing the *design gap* is essential particularly for semiconductor companies fully or partially following the fabless approach.

The last disruptive productivity increases have been achieved by semi-custom design, RTL synthesis, and so-called "IP-reuse". Besides automating the construction of the lower level implementation, these approaches heavily rely on reusing pre-implemented pieces[1]. Although each of these approaches brought a big leap in productivity, the general benefit of reuse is often overestimated – an insight acquired in [4] as well. One reason for why TLM has not brought productivity increases may be that it does not define one agreed abstraction as RTL (synchronous design, time discretization) and Gate-Level (digital design, value discretization) do.

A measure proposed to further increase design productivity is the generation of code from more abstract descriptions. Ecker et al. reported a 20x productivity increase in special design tasks and up to 3x higher productivity in design implementation from specification freeze to tape-out through the use of Metamodeling and code generation [6]. The replacement of ad-hoc script based approaches by a more structured approach and the link to formalized specification data have been mentioned as key to this improvement.

Generation as part of a completely new approach of designing chips has been postulated by Nicolic [17] and Shacham et al. [20]. Generators should not only be used to make simpler and more efficient designs but also to generate different alternatives, thus to enable exhaustive architecture analysis. The use of generators instead of models has been claimed by Bachrach et al. [3] as well. Bachrach is developing Chisel, a so-called hardware generation language (HGL). He has demonstrated a code reduction by a factor of 3x in several application examples, when comparing Chisel models and Verilog code. While Chisel provides a very compact and generation centric RTL notation, micro-architecture pattern[13] go a step further and generate via an intermediate called DHDL complex pipelined micro-architecture patterns.

Interestingly, IP-XACT [10], the standard for IP integration, was prepared from the beginning to include generators and generator chains in the integration process. Unfortunately, no measure was described to formulate and build generators efficiently.

Besides controlling and configuring generation, building generators is the most time-consuming effort to enable automation. With Model-driven Architecture (MDA) [18, 22], the OMG offers a vision for target code generation from specification via a set

[1] Please note that beyond functional mapping and time abstraction to clock cycles, a big contribution in RTL's productivity results from mapping operators such as "+" to predefined structures such as adders.

of models and a set of transformations, each deriving a model from the adjacent model. For further reading, it is important to note that we refer to *model* in terms of SW engineering and Metamodeling. A *model* is an instance of a Metamodel and a Metamodel is a class diagram-like notation[2] of the model's structure. In other words, a model specifies a set of entities – or things – their relations and their attributes.

Another challenge in making generators – or as part hereof translators – is the consideration of different Models-of-Computation (for a definition, see e.g. [7]). The generator developer does not only have to think in terms of generated design target but also in terms of simulation artifacts of the target languages that are eventually synthesized to hardware. Examples for simulation artifacts are blocking and non-blocking statements in Verilog, delta races caused by clock gates in VHDL, and proper `'X'` handling. Further examples are the type inconsistency between std_logic_vector and std_ulogic_vector in VHDL and additional code artifacts that are necessary as it is prohibited to read output ports.

In this contribution, we present an approach that follows the MDA vision postulated by the Object Management Group (OMG). MDA describes code generation via a chain of models in order to simplify and improve the construction of generators. In the next chapter, we describe the MDA idea in more detail and explain how we adopt it to digital hardware design. After that, we describe the components of our framework and the individual models and metamodels. Here, we provide simple examples for transformations and application sketches. First, we describe the Model-of-Things a specification related and design independent layer. Next, we introduce MetaRTL, a (digital hardware) Model-of-Design (MoD), acting as key model of our proposal. This MoD not only simplifies and accelerates generator construction by providing a more abstract target. It also helps to avoid thinking in terms of simulation semantics and artifacts. Here, we also show how MoDs can be efficiently built using a Templates-of-Design (ToD). To make transformation to RTL code complete in order to be able to utilize state-of-the-art design flows, we introduce as a third model the Model-of-View (MoV) an abstract syntax tree like structure. The application of the proposed methodology for generating different flavours and styles of code for different processors and the benefits of the approach are discussed next. After introducing the general framework and our overall approach, we describe real-world application examples and use them to compare our hardware generator construction method to related generation and modeling approaches. Eventually, we describe related work in the field of hardware generation and construction approaches and evaluate how our approach competes.

2 Model-driven Architecture

2.1 The OMG Model-driven Architecture Vision

Model-driven Architecture (MDA, [18, 16, 22]) is an OMG vision of the future of software design that popped up a bit more than 15 years ago. MDA approaches a growing productivity gap that also burdens software engineering by fostering code generation.

[2]There are other notations as e.g XML schema definitions or entity relationship diagrams. For an overview see e.g. [5].

Instead of using simple model-to-code generation, MDA involves some intermediate steps before the final code is generated.

The first version of MDA introduces three kind of models, namely a Computation Independent Model (CIM), a Platform Independent Model (PIM) and a Platform Specific Model (PSM), where:

CIM is the most abstract one and closest to specification. It considers neither detailed algorithm implementation nor architecture.

PIM already defines the architecture – and therefore also the implementation but avoids details of the platform.

PSM is platform dependent and closest to the target code, also referred to as view. From this model, the view is generated.

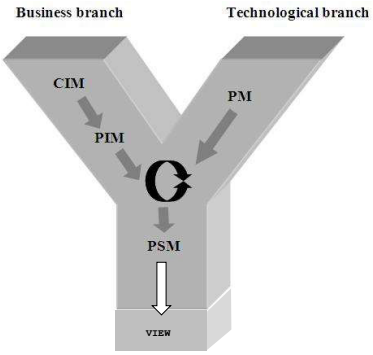


Fig. 1: MDA pictured as Y-Chart from [14]

Figure 1 shows this dependency in a Y-Chart for which MDA is well known. In addition to the aforementioned models, this figure also shows a Platform Model (PM), which contains the details of the target platform.

A remaining question is "what is a platform?". In the software world, a platform is 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.

Before we describe our adaptation of MDA to digital hardware, we will first analyze some limitations of the initial MDA definition. First, the definitions of CIM and PIM are a bit vague, especially when it comes to the definition of functionality. Second, the agreement on 3 intermediate models is often at issue and it is questionable why there aren't 1, 2 or 4 layers of models instead, depending on the application domain and the difficulty of transformations. The third and last issue relates to the the position of the Platform Model in the Y-Chart. This model is a description of the platform a PIM is mapped onto. The level of abstraction of this description is thus comparable to that of the PSM and it is depicted too "high" in the Y-Chart, since it is closer to the PSM's level of abstraction.

The problem with the modeling layers is relaxed in the second version of MDA released mid 2014 [18]. Here, a characteristic of a transformation is that it translates a PIM to a PSM. A sequence of any number of transformations is chained together to finally generate the view. At 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. The target code is then compiled together with packages and libraries, etc. to a product, suitable to run on a computing platform[3].

2.2 Model-driven Architecture for Hardware Development

Figure 2 shows our approach to adopt MDA for digital hardware generation. It conceptually follows the 3-level model of the first MDA proposal but has important enhancements to support hardware design. For instance, it introduces terms for the various hardware related models that are involved, each being more closely related to the purpose of the model they describe:

MoT The Model-of-Things corresponds to the CIM since its intention is to formally capture data from requirements and specifications. By doing so, the MoT defines things, their attributes and the 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/structure are used. An example for a MoT is the instruction set of a CPU as discussed in Section 7.1.

MoD The Model-of-Design corresponds to the PIM since its goal is to define the architecture using designer's terms. The MoD and its Metamodel are the core components of our methodology. Broadly speaking, a Model-of-Design describes the design on RT-level. It differs from modeling languages based on event-driven semantics as it does not cover any simulation or synthesis artifacts of the views we target with our approach. We detail the MoD and its Metamodel in Section 3.

MoV The Model-of-View corresponds to the PSM since it is the least abstract model with a straightforward mapping to the target view. It is more or less an abstract syntax tree defined by an abstraction of the target code's grammar. In future, we might extend this model with some target architecture related information as mapping pragmas or target platform specific code snippets. The Model-of-View layer is detailed in Section 6.

For each of the MDA-layers, we define Metamodels. These metamodels constrain the valid models of the modeling layers. There are several Metamodels on the MoT layer and on the MoV layer respectively. For the MoD layer, there is exactly one metamodel which is related to the RTL and synchronous design abstraction. For targeting other implementations such as analog hardware or firmware, additional Models-of-Design are needed. Every specification formalized in a Model-of-Things instance is

[3] 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 [12]. In other contexts, the term platform is used for a product family which addresses one field of application in various configurations.

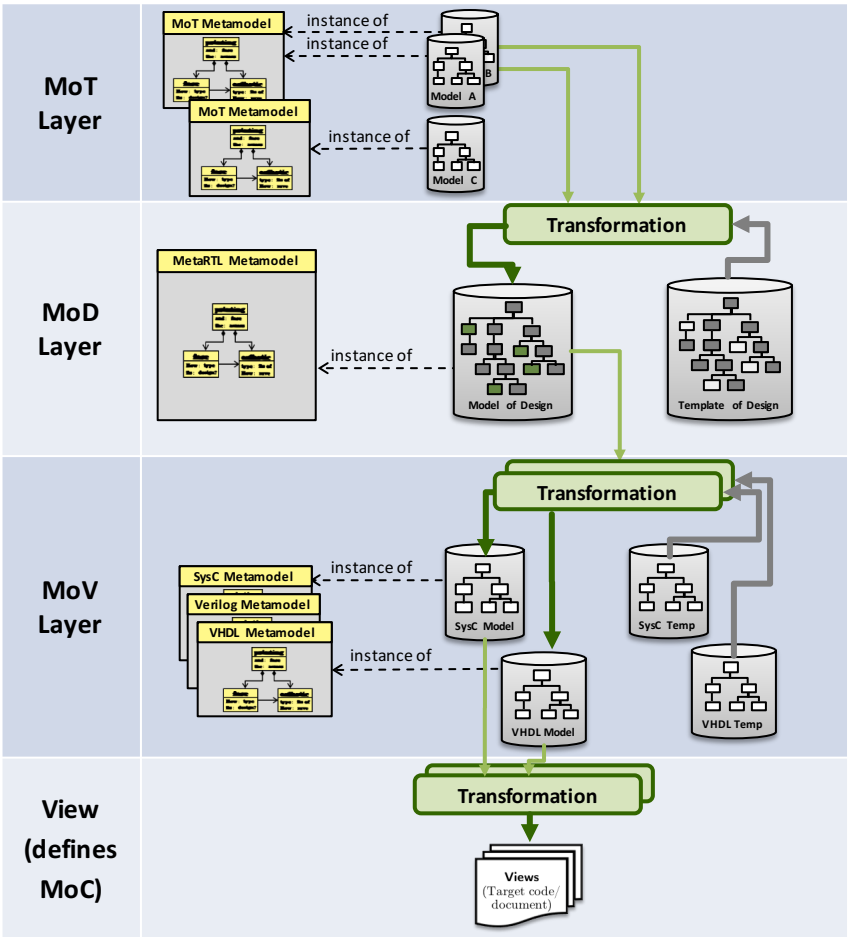


Fig. 2: MDA for Hardware Generation

transformed onto a set of RTL components part of a Model-of-Design. On the Model-of-Things layer, several different metamodels 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. On the Model-of-View layer, a different Metamodel is necessary for every view language, while two different views of the same language use the same Metamodel.

In addition, Figure 2 introduces the Templates-of-Design (ToDs, see 5.2) and Templates-of-View (ToVs, 6). They relate to the Platform Model (PM) in MDA's Y-Chart representation from Figure 1. They are pictured right of the MoD and MoV respectively, illustrating that they are on the same level of abstraction. These templates are designed for a certain Metamodel on the respective layer of abstraction and therefore work on the API provided for that Metamodel. The Template-of-Design (ToD) describes the targeted architecture and directly instantiates Model-of-Design (MoD) items. For this instantiation, it can utilize data from several Model-of-Things (MoT) instances. It is thus possible to intuitively integrate configurable and non-configurable components. The result of the Template-of-Design execution is one static Model-of-Design instance. This instance can be transformed on the Model-of-Design layer and can further be transformed into a Model-of-View instance using the Template-of-View (ToV). Similiar to the ToD, the ToV relates to the structure of the target code. It instantiates elements of the MoV, which are more-or-less nodes of the abstract syntaxt tree of the target view.

Similar to MDA 2.0, our adoption of MDA to hardware does not insist on exactly three modeling layers. Instead, the MoV can be omitted and the view can be directly generated with a template engine from the MoD. Further, several MoTs can be used to build the MoD and there may be transformations between MoTs before the MoD is constructed.

Our adoption of MDA realizes the two key aspects for productivity improvement already mentioned in the introduction: automation and reuse. Automation is established by transformations and reuse is enabled by re-using models, their Metamodel-based definitions, and through reuse of existing transformations and view generators.

3 Framework

This section introduces the framework we utilize for our Model-driven Architecture approach. For this purpose, we first introduce Infineon's proprietary metamodeling framework. We then show auxiliary elements which are utilized across different Metamodels.

3.1 The Overall Framework

Our metamodeling framework uses a subset of an UML class diagrams, extended with some features from XML Schema to specify Metamodels. Substantially, objects, their attributes and various kinds of relations between them can be specified. The Metamodel in turn is defined by a Meta-metamodel, which is also used to relate and combine Metamodels as well as to generate Metamodels from other descriptions like an XML Schema

or EBNF grammars. The Meta-metamodel uses the same notation as the Metamodel, i.e. is self-defined.

The Metamodel can be captured in a textural and a graphical way. The framework generates an infrastructure for handling the Metamodel and its models. This infrastructure supports the generation of extendable APIs, code generation for persistent storage of the models and tool frame generation. The generated tool frame supports joint handling of various models, creation of models, transformations between the models, and generation of views. The framework is written in Python, uses Python for manually coded and generated transformations and takes advantage of numerous Python libraries and tools including the Mako template engine for a direct Model to View generation.

Python is not only used because of the already mentioned libraries and rich features such as object-oriented programming, aspect-oriented programming utilizing introspection and functional programming. Equally important is its low adoption barrier. We observed that Metamodeling frameworks such as the "Eclipse Modeling Framework" (see [21]) are more powerful but very complex and thus very hard to learn and adopt for hardware engineers. With a Python-based approach, we managed to lower this barrier significantly.

The framework makes the development of MDA flows easy by generating the already mentioned tools. Both Template-of-Design (ToD) and Template-of-View (ToV) are developed as Python code utilizing the generated API to fill their target model and to access the more abstract source models. The models reside as data structures in the generated framework tools. These data structures can be read from and written to XML files.

The uniform structure of the modeling flow also makes it easy to understand and use the Metamodels. Thus, the different levels of detail resulting from different levels of abstraction from MoT to MoV can be easily handled. Descriptions can be made in a uniform way, transformations are less painful and there are fewer inconsistencies.

Although the Metamodel enables the described tool building framework, Metamodeling is still mainly a modeling activity since it structures and defines the objects, attributes and relations as abstractions of the design space.

3.2 Auxiliary Metamodels

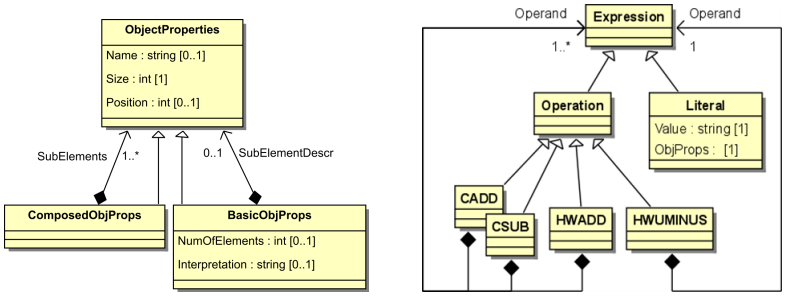
The above-mentioned Meta-metamodel-based features of the framework make it easy to structure Metamodels and to reuse predefined Metamodels. We currently support two auxiliary Metamodels which are used on different layers of our MDA approach: the Type Metamodel and the Expression Metamodel.

These two Metamodels are used on the Model-of-Things and the Model-of-Design layer. They are intended for use wherever Metamodels contain elements describing objects and behavior. It is for example necessary to describe object sizes (e.g. the sizes of registers) on both the Model-of-Things layer and the Model-of-Design layer. The utilization of the same auxiliary descriptions across different metamodels has two 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, ob-

jects and expressions can be simply compared and copied between different MoD and MoT instances.

The Type Metamodel follows the idea describing data types 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. Types can be named to ease 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 3a shows a UML representation of our type Metamodel.

The Expression Metamodel defines a tree of operators, with both classical operators from programming and specific hardware operators such as add-with-overflow or select. Figure 3b contains a UML representation of a subset of our Expression Metamodel. The expression hierarchy is defined by the location in the tree. There is no defined operator priority or counterpart to brackets for prioritizing operators. For each operator, the size of the output depends on the size of the inputs. Further, relations between the inputs are specified. There is however no size limit as a such. Leaf nodes can be binary or numeric literals in any combination or references to objects with properties defined by the type model. The Expression Model is a special case of a Dataflow Model. The Dataflow Model is a directed graph with the same operators as the Expression Model. It is different from the Expression Model as any operator output can act as input to multiple dataflow operators.



(a) Type Metamodel (b) Expression Metamodel

Fig. 3: Auxiliary Metamodels

These two Metamodels are used on the Model-of-Things and Model-of-Design layer. They are intended to be used whenever Metamodels contain elements describing objects and behavior. It is for example necessary to describe object sizes (e.g. the sizes of registers) on both the Model-of-Things layer and the Model-of-Design layer. The utilization of the same auxiliary descriptions across different Metamodels 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, ob-

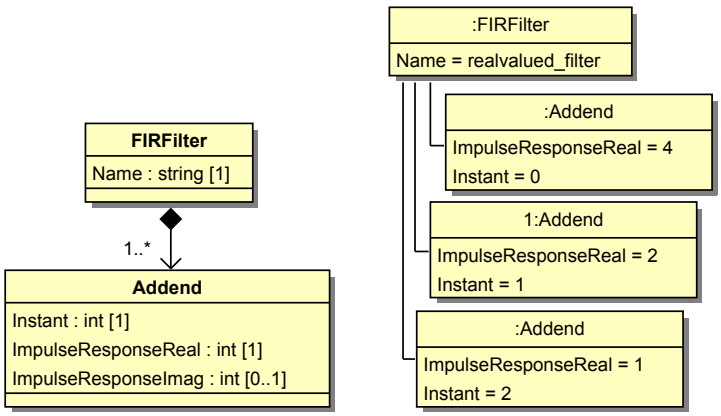
jects and expressions can be simply compared and copied between different MoD and MoT instances. Third, the mapping of types and expressions to the target view must be implemented only once and can be reused for all implementations. Latter holds for both possibilities for view generation, the direct code generation from MoD using a template or having a MoV in between. The first alternative produces quite rapidly results but becomes complex if some flexibility or formatting is needed in code generation. The second alternative requires thinking in terms of grammar blocks but keeps the burden of formatting to a one time effort and makes it simpler to make code generation more flexible.

4 The Model-of-Things Layer

To ease understanding of the Model-of-Things concept, this section provides a simple sample Model-of-Things along with its Metamodel. We utilize this example to illustrate how the Template-of-Design generates a Model-of-Design instance in Section 5.2. A more complex example is sketched in Section 7 of this contribution.

The Metamodel in Figure 4a constrains all digital filters. Our Model-of-Things instances can thus capture digital filters which adhere to a recurrence relation $y[n] = \sum_{i=0}^{N} b_i \cdot x[n-i]$. Figure 4b shows a sample model that adheres to this Metamodel. The attributes `ImpulseResponseImg` does not show up in the model since it is not needed. This complies with the Metamodel since `ImpulseResponseImg` has multiplicity optional, i.e. `0..1`.

This model describes one 2nd order instance a filter that has the recurrence relation $y[n] = 4 \cdot x[n] + 2 \cdot x[n-1] + 1 \cdot x[n-2]$.



(a) Metamodel (b) Model

Fig. 4: Model-of-Things metamodel and sample model of simple FIR filter

Despite the simple structure of this example, a key characteristic of our 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 7). In this case, the multipliers might be replaced by wires treating the coefficients as constants for the cost of loosing 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 both equally valid microarchitectures considering a specification that only describes the transfer function of the digital filter and does not contain any performance constraints such as throughput.

As this example illustrates, there is not necessarily a one-on-one correspondence between elements in the Model-of-Things and elements in the Model-of-Design. As we utilize a flexible Python-based transformation approach to generate the Model-of-Design, it is feasible to first derive the coefficients b_i for a digital filter. It is for example possible to use a Model-of-Things that specifies intended frequency-domain characteristics and maximum deviation from these characteristics. The large set of scientific computing libraries available allow to perform the necessary computations as part of the transformation between Model-of-Things and Model-of-Design, without requiring external tools or manual work. To be more concrete, Python's `scipy` provides a package `signal` which supports the computation of filter coefficients using the functions `firwin` and `firwin2`.

5 The Model-of-Design Layer

5.1 The MetaRTL Metamodel

Figure 5 shows MetaRTL, the Metamodel of the Model-of-Design (MoD). It defines, how the design structure is described in the MoD. MetaRTL consists of a root node and components with realization alternatives: Structure, Primitive, Dataflow as well as – in preparation not shown in Figure 5 – Controlflow, StateMachine, and Lookup-Table.

The last four of these are configured with other Metamodels describing their implementation. For example, the Dataflow realization of a component is configured by an instance of the Dataflow Metamodel that is linked in the DataflowBlock. The four models can be clocked, un-clocked, with or without internal delay. If they are clocked or have an internal delay, a clock and reset is associated automatically.

The Primitive and Structure realization alternatives are more flexible. They support the working mode of concept engineers and architects who often avoid the specification of port names or sizes, if they can be derived from elsewhere, e.g. from a sketch, or are common sense (as e.g. ports of a register are named q and d whereas q and d must have the same size).

There are special kind of ports like clock or reset. These ports are automatically connected to the port closest in the hierarchy, and labeled the same special kind of ports. Of course, all connections and ports can be made explicitly and with accurately specified sizes. Then the connectivity resolution only checks the type – i.e. size – consistency of the connections.

Further, the connectivity specification supported by the Metamodel can not only connect a port with another port but can also connect a port with a component and can

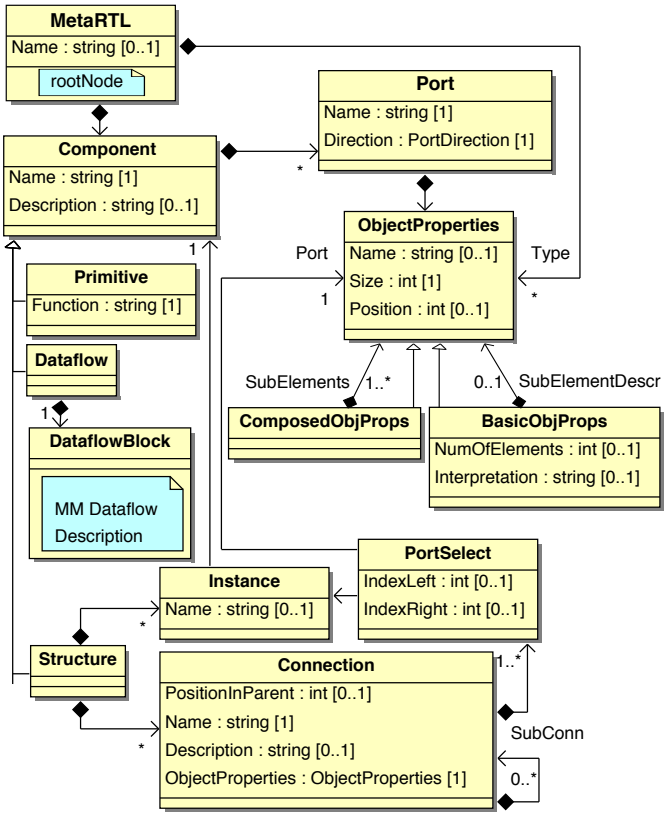


Fig. 5: MetaRTL: MoD's Metamodel (simplified)

connect a component with a component. The connection can be associated with a set of properties that help the resolution mechanism to resolve the intended port or the size of port and connection.

Also setting of component parameters can be resolved from the connectivity. Thus, setting a connection to a reset port can determine whether a register has a reset behavior or not. Of course, all parameters can be also set explicitly in the instantiation. The resolution strategy is described in more detail in 5.4. Primitives such as registers, adders, multiplexers, decodes, etc. have methods that determine the final parameter setting and port layout.

To give some examples, in the register primitive, the size of q and d can be propagated forward and backward, i.e. the size of the data output q can be derived from data input d and vice versa. Connections to q and d can be made explicitly or distinguished by read/write or in/out property of the connection. Also a name, let's say a, can be specified. Then the register has slightly different ports, namely q_a and d_a in this case. Using this mechanism, a register may be configured to have more than one data input and output pair.

Propagation of sizes is limited when instantiating e.g. an adder. Here, the size of the output is the maximum of the size of the two inputs plus one. Thus, only a forward propagation of the sizes is possible.

Any number of connections can be applied to the output of the decoder. Then, a port is created for each of the connections. Further, each connection can have a property specifying at which select value it is enabled. This property, which can be also computed automatically, is used further on to define the behavior of the decoder and the size of the select signal.

5.2 Templates-of-Design

The Template-of-Design (ToD) acts as a blueprint or generator for instances of the Model-of-Design. It contains the instructions to build different instances of this model and thus different digital designs. It is called template since it pulls information from the more abstract MoT and since it looks almost like an HDL netlist description when a fixed architecture is described. In our implementation, the ToD is pure Python code and is not to be confused with the templates processed by template engines such as Python's Mako for target code generation.

The HDL netlist-like style (see Figure 6) is enabled by object constructors and attribute setters and getters generated for the MetaRTL Metamodel. It is important to note that the Python code is not the hardware model. Instead, it constructs the hardware model. When Lines 2, 3 and 4 are executed, new components are instantiated in the Model. The loop in lines 7-9 creates a new connection every time it is run.

```
1 width = 32
2 alu_input_1 = Connection(ObjProps=SimpleObj(width), parent=alu)
3 alu_input_2 = Connection(ObjProps=SimpleObj(width), parent=alu)
4 alu_mux = Mux(Name='ALU output Mux', parent=alu)
5
6 for unit in alu.units:
7     alu_input_1.connect(unit.Ports['input_1'])
8     alu_input_2.connect(unit.Ports['input_2'])
9     unit.result_port.connect(target=alu_mux, ConnName=unit.name)
```

Fig. 6: Code snippet from a ToD using named method parameters

The power of the Python approach can be illustrated in following example. If a specific coding style for connections must be met, the parameter setting `ConnName=unit.name` can be modified with a small overhead e.g. to `ConnName=unit.name+"_s"`

The code snippet is taken from a small ALU component which contains several functional units which process two input arguments (connected to `alu_input_1` and `alu_input_2`). For each functional unit (stored in `alu.units`), the code connects the output port to the `alu_mux` component (Line 9). Moreover, the input ports of the

functional unit are wired to the connection objects `alu_input_1` and `alu_input_2` in Lines 7 and 8.

As the ToD is pure Python code, it provides the full flexibility of the Python environment for importing and versioning. If parts of different MoDs are similar, their ToDs can share the same code. This means that reuse is done on ToD-level and MetaRTL must not support the concept of a library of (reusable) components. This significantly reduces the complexity of MetaRTL and any transformations from its MoD-level to different MoVs.

5.3 Model-of-Things instances for Template-of-Design construction

In addition to describing circuits in the static HDL netlist way, the complete feature set of Python can be utilized. This provides more flexibility to define the architecture. Section 4 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 our 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. We describe a scenario where we could successfully utilize this in Section 7.2.

To illustrate the concept 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. The Template-of-Design is built for the Metamodel in Figure 4b and will construct MoD instances for all models of this Metamodel. Figure 7 contains a block diagram of the circuit it creates for the sample model in 4a.

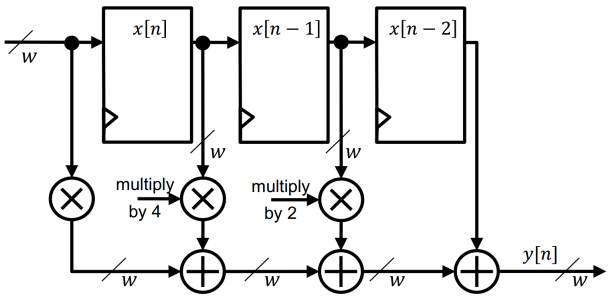


Fig. 7: Block diagram of MoD generated from the MoT in Fig. 4a by ToD in Fig. 8

The Template-of-Design source code is pictured in Figure 8. When executed, line 33 of this instance adds a `FIRFilter` instance to the Model-of-Design. The construction of this FIRFilter takes place in the constructor in Lines 3-31. First, the Ports of the filter are defined. We only define the ports `data_in` and `result` here and omit 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.

```
class FIRFilter(Structure):
  def __init__(self, firMoT, parent):
    super(FIRFilter, self). \
      __init__(Ports=[{'Name': 'data_in', 'Direction': 'IN'},
                      {'Name': 'result', 'Direction': 'OUT'}],
               parent=parent)
    current_delay, sum_conn = 0, None

    current_conn = Connection(self['data_in'])
    for addend in sorted(firMoT.Addends,
                         key=lambda x: x.Instant):
      while current_delay < addend.Instant:
        reg = Register()
        current_conn.connect(reg.In)

        current_conn = Connection(reg.Out)
        current_delay += 1

      mul = HWMUL(Constant=addend.ImpulseResponseReal)
      mul.addIn().connect(current_conn)

      instant_out = Connection(mul.Out)
      if sum_conn is None:
        sum_conn = instant_out
      else:
        adder = HWPLUS()
        adder.addIn().connect(instant_out)
        adder.addIn().connect(sum_conn)
        sum_conn = Connection(adder.Out)

    sum_conn.connect(self['result'])

fir_MoT_1 = ... # instance of a FIRFilter Model-of-Things
myFilter = FIRFilter(fir_MoT_1, toplevel)
```

Fig. 8: Template-of-Design example for generation of n-th order FIR filter from Model-of-Things

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 an ascending order. Lines 12-17 contain a while loop including loop body which is executed to insert delay registers. In our sample MoT, we use 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 futher 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.

5.4 Connectivity resolution

The ToD can be kept simple, since our framework has a powerful resolution mechanism to do the final connection. Here, ports are created, connections of default ports, and size computation is done.

The mechanism is kept simple, since the instantiated components either utilize generic methods to compute component port information from other ports or provide special ones. The resolution algorithm calls these methods if new information about a connection has been computed and takes the new result of the method to refine information about connections. This is repeated as long as new information either about component's ports or connections can be found. At the end, either all components are fully configured – including availability of all ports – and all sizes are known, or an error message is filed summarizing the non determined items. More information must then be placed in the description of the ToD.

So far, resolution depends on component local implications at ports and connections. Potentially, the use of a formal SMT solver might resolve some more items. First analysis however showed that it is not worth the effort, since the number of additional resolutions is small, complexity issues pop up soon and the reports aren't that easy to interpret.

6 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. Reasonable target platforms are HDL code for synthesis

or simulation purposes. The Metamodel of the Model-of-View is tailored to the language of the target view. Two views both targeting Verilog code utilize the same Metamodel while a SystemC view would require a different Metamodel. When the same view language is used to generate code for different target platforms, the transformations are changed. A Verilog model for ASIC synthesis and a Verilog model targeting an FPGA is built from the Model-of-Design with two different transformations[4], while both MoV models utilize the same Metamodel. As those target view languages have their inherent Model-of-Computation (MoC), any Model-of-View also inherently defines the MoC.

The Metamodel of the Model-of-View layer specifies a target view similar to how Extended Backus-Naur Form (EBNF) notations describe the formal grammar of a language. The metamodel constrains the possible Model-of-View instances so that all legal instances will translate to grammatically correct views. Although there is a straightforward correspondence between the Model-of-View and the generated target views, the MoV-layer of our MDA inspired approach provides an important abstraction: it allows the developer to think about the view he wants to generate, without worrying about necessary formatting or indentation, as these are not part of any Model-of-View. Formatting and indentation are specified independent 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.

6.1 Automated View Generation from Model-of-View Instances

When developing the framework for our Model-of-View layer, we put special emphasis on automation. Figure 9 sketches this generation. We use a single EBNF-like description that contains information about the grammar of the target view as well as formatting and indentation of the generated code. This description is called View Language Description (VLD). It is utilized to generate the majority of the necessary MoV-layer components. First, the Metamodel of the Model-of-View is built and based on this Metamodel, the metamodeling framework provides the API which used to read and write Model-of-View instances.

Furthermore, the code generation step, building the target views from any Model-of-View instance is completely automated. The general algorithm is based on tree-traversal, where leaf nodes produce view code that is encapsulated by view code produced by their parent nodes. Eventually, the view code generated for the root node, containing all code of its child nodes is printed to a view.

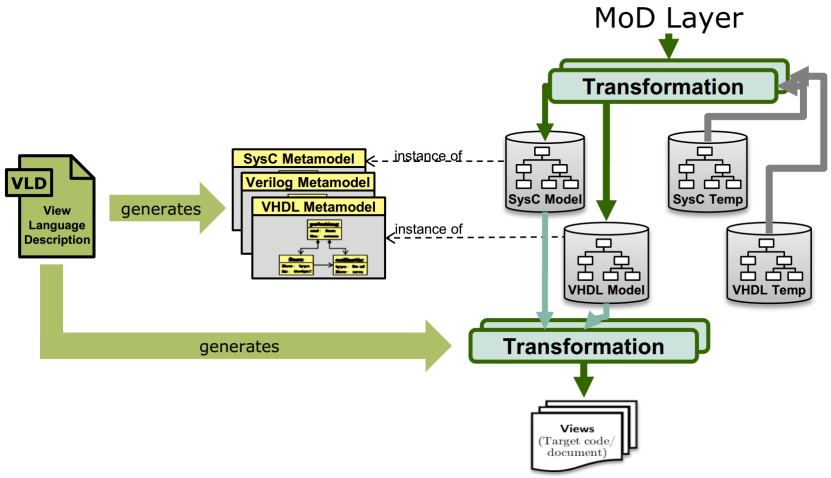


Fig. 9: View Language Description based Metamodel-of-View generation

```
1 Entity ::= 'ENTITY ' <Name> ' IS\n'
2                [Ports]
3            'END ' <Name> ';\n';
4 Ports  ::= $indent('\t')$('PORT(\n'
5                $indent('\t')$(+Port%[0:-2]: ';\n';
6                                        [-1]  : '\n'  %+) ');\n');
7 Port   ::= <Name> ' : ' <Mode> <Type>;
```

Fig. 10: Simplified Snippet from View Language Description of our VHDL MoV [19]

6.2 View Language Descriptions

Figure 10 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. Similar to EBNF, our description consists of a list of production rules where each of those rules in turn consists of a set of terminal or non-terminal symbols. The main goal of EBNF is to describe formal language grammars and thus to provide the rules for distinguishing grammatically correct code of a certain formal language from wrong code. As outlined in the previous section, our VLD has slightly extended goals: we utilize it to generate the Metamodel of the Model-of-View. This introduces three new requirements for our 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

[4] Of course, the different transformations share several sub-transformations. However, we are analyzing, if two transformations or one transformation to the model of view is best, and if the one transformation uses parameter dependent code or if the one transformation is finished, and a further transformation is performed on the Model-of-View

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

Fig. 11: VHDL entity declaration from VHDL'93 (see `tams.informatik.uni-hamburg.devhdltoolsgrammarvhdl93-bnf.html`)

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, however semantically irrelevant should not be part of the Metamodel. An example for 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 independent from the others.

2. While EBNF is structured to easily and efficiently build parsers, VLD shall provide a Meta-Model to construct code. So, an entity rule directly contains `ports` (see Fig. 10 in line 2) and `generics` as well as `library clause` and `use clause` (not shown in the simplified VLD example in Fig. 10). The VHDL grammar of the entity declaration shown in Fig. 11 is structured quite differently. So, `library clause` and `use clause` are defined in `context clause`, which is used in `design unit`. This makes it easy to reuse code in the parser, would require however that additional nodes have to be generated in the MoV, which reduces convenience of the approach.
3. It has to describe the indented 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 indendation. From parser point of view, any number of identical views with different formatting will map onto the same Abstract Syntax Tree. From generation point of view, one formatting options shall be choosen when generating the code.

To meet those requirements, we introduce serveral formalisms into the VLD. When the Metamodel for a VLD format is generated, we add a Metamodel class for every production rule of the VLD descriptions. For every non-terminal symbol a rule consists of, we add associations to this 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 those 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, we introduce a `+...+` symbol into our VLD. The use of this symbol will create 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, we introduce a further artifact into our VLD. This artifact is used in the `Ports` rule of the VLD in Figure 10. Here, the `%%` notation 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 our format is the distinction between non-terminal symbols that create attributes in the Metamodel (encapsulated by `<...>`) and symbols that create compositions (not encapsulated `<...>`). We further use the `:` operator 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.

A naïve approach to formatting target code in a nice way is inserting terminal strings containing whitespaces into the VLD. The problem of 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 architecture level, inside a process or inside any number of nested conditionals, each requiring a different level of indentation. Our 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 and work by post-processing code that has been generated. We provide a set of predefined directives for correct indentation of code, line breaks at certain line widths and correct alignment of neighboring lines.

To give an example: `$indent('\t')` in Fig. 10 in line 4 ensures an additional indent of keyword `PORT` and the following parts by a tabulator. Similarly, `$indent('\t')` in line 5 ensures, that each port item is indented one tabulator further.

7 Application and Results

To demonstrate the general feasibility and the advantages of our MDA approach to digital hardware design, we are implementing a CPU subsystem generator. The input models for our flow are Model-of-Things instances which are utilized to generate different aspects of the target views. Our generation system relies among other things on a model capturing the instruction set of the CPU core part of the subsystem. The Metamodel of this model is called MetaRISC and suited to describe different RISC ISAs. We describe this Metamodel and how it is used by the CPU core generator in the following section. Our approach further relies on models that describe aspects other than the instruction set of the CPU. These models and their Metamodels cover peripherals such as the timer and interrupt controller and custom processing peripherals similar to the filter example in Section 4.

7.1 Model-of-Things of the CPU core defined by the MetaRISC Metamodel

MetaRISC is the Metamodel used to describe the ISA of CPUs our core generator can handle. The key requirement for identifying Metamodels for the MoT-layer is to formalize the possible MoT instances as much as necessary, yet to permit reuse and automation

over a wide range of specifications of the same category of hardware component. In our case, this formalization constrains our ISA descriptions to RISC instruction sets. We can e.g. use it to describe the MIPS-II, RISC-V and ARM Cortex ISAs, is however not limited to these kind of instruction sets.

Conceptually, MetaRISC describes the programmer's view on the CPU core. The metamodel therefore contains elements for architectural state, instruction behavior and for instruction encoding. The architectural state of the core describes every state element visible to the compiler. Examples for these elements are the program counter, state flags and the memory and register file of the core. The description of the instruction behavior covers how the execution of any instruction modifies the architectural state of the CPU core. The instruction encoding covers both the detection of the correct instruction and the decoding of immediate parameters and register addresses encoded in the instruction.

For both the description of the architectural state and the description of the behavior, we rely on the auxiliary Metamodels described in Section 3.2. This is a key aspect that helps us transferring the Model-of-Things specification to Model-of-Design microarchitectures implementing them.

Before discussing the ToD, it is important to note that the behavior described in the MoT is not used to synthesize the implementation directly. Instead, it defines a reference behavior, the MoD being constructed by the ToD (without using the behavior) has to comply to. Therefore, the behavior can be used - amongst others as documentation generation - to generate a testbench or properties that validate the generated view.

7.2 Assembly of a Template-of-Design for Micro-Architecture Generation

The most substantial and elaborate part of the MDA flow is the assembly of a Template-of-Design that constructs Model-of-Design instances which describe an implementation of the formalized specification captured in the Model-of-Things.

From a birds-eyes view, a generic ToD pieces together parts of the Model-of-Design so that the functionality described by the Model-of-Things is provided. This is a relatively straightforward process for components such as the FIR filter described in Section 5.3.

For the CPU core generator that is implemented here, the ToD provides a pipelined architecture that implements the CPU's Instruction Set. Again, the bulk of this construction is a rather straightforward instantiation of MetaRTL structures and instances that are part of these structures as e.g. introduced in [8]. These for example define the overall pipeline structure and the functional units in the ALU of a CPU. The Template-of-Design provides automation for these tasks as sub-components may be instantiated or not depending on the MoT. For example, functional units of the ALU may be instantiated only if there are instructions that rely on them.

For some sub-components of the CPU core however, a large dependency between their inner structure and the Model-of-Things exists. Decoding of instruction arguments and implementation of Control Unit have high dependency on the encoding specified by the ISA Model-of-Things. From the Model-of-Design's perspective, a control unit is a decoder which can decode up to n-of-n out wires for a certain input combination. To provide this description, the Template-of-Design has to fulfill the behavior described in the MoT.

Here, Python as the framework's underlying language is very useful: it allows to use a scripting approach for fast and creative solutions. As both source and result data are embedded into the formalized MDA stack, re-usability is not sacrificed.

The construction routines which were built as part of the ToD for example allow to create a testbench which uses both the models of instruction behavior and a skeleton of the generated CPU to find working control signal vectors for each of the instructions part of the ISA. For our simplistic architecture, it was sufficient to brute-force all possible combinations of control signals for an instruction to find the combinations that yield a behavior consistent with the ISA description. More sophisticated methods such as functional equivalence checking with formal verification tools might be an option we follow in the future. Both approaches can be carried out from within the MoT-to-MoD transformation thanks to the flexibility of Python.

7.3 Results and Discussion

The presented hardware MDA approach was applied to a MIPS2 integer instruction set and the RISC-V [1] integer instruction set with the compressed instruction set extension being mapped to 2-, 3-, 4- and 5-stage pipelined microarchitectures. To evaluate the benefits of our multi-level transformation approach, we compared our hardware MDA approach to a direct MoT-to-code approach (utilizing a Python-based template engine called Mako). Our comparison showed that the MDA approach requires about a factor of 10x less code compared to generating VHDL with Mako templates. They key part of this reduction comes from the high re-usability of ToD code compared to template code and partly also originates from a higher expressiveness of Python and the automation in the transformations.

The Metamodel based specification of MoT and MoD provides an intuitive interface to models and gives a low entry barrier. The APIs and object oriented approach also supports generic template components. For example, we implemented a pipeline template which automatically generates the registers and connections between different pipeline stages. This pipeline template was developed in a one-time effort, can however be reused for other designs. Everything else just works, shielding designers from thinking in simulation semantics allowing them to focus on the challenging aspects of the design task.

8 Related Approaches

There are many academic and commercial approaches generating RTL code from UML or other graphical notations. Although these approaches use Metamodels, none of them links to specification and has a hardware design aware model.

The idea of focusing on design – i.e. in our approach on the Model-of-Design (MoD) – and not on simulation semantics is not new. For example, UDL/I [9, 2], the *unified design language for integrated circuits definition*, first targets the design intent and only as a second step the execution of its models. UDL/I was developed and standardized in the early 90ties by Japan Electronic Industry Development Association JEIDA, almost in

parallel with Verilog and VHDL. It did however not make it to a wide distribution. Design aspects instead of simulation aspects are also the focus of UPF [11], since it permits the insertion of things like level-shifters or isolation cells without explicitly providing a specification of their simulation characteristics. Digital design related descriptions are also supported by EDA tools, either through the use of generic components in graphical editors or as intermediate step in the RTL synthesis process. In comparison to our approach, these description styles lack an explicity underlying Metamodel, support for hardware generation, an explicit, automated link to specification and powerful measures to describe connectivity and architecture alternatives.

The FIRRTL (Flexible Intermediate Representation for RTL, [15]) format is a textual intermediate that is used when Chisel hardware generators [3] are compiled to Verilog. Similar to our Model-of-Design model, FIRRTL models describe design items, can use powerful features to specify instances and connectivity in a simple way, provide hooks for further optimization and can be translated to an RTL view (Verilog).

FIRRTL offers the possibility to hook optimization routines and transformations. Chisel with FIRRTL thus shares the ideas of generation, intermediate transformations and design thinking with our approach.

It differs from our approach as it only defines an intermediate language and not an intermediate model, i.e. it has no explicit Metamodel. Also, FIRRTL does not share a common type and expression system with its high-level counter part Chisel. FIRRTL further does not provide a link to MoTs or specifications and a ToD approach allowing to merge (micro-)architecture specification with methods deriving and processing data from MoTs. In other words, our approach utilizes one common infrastructure to automate design from specification to implementation while Chisel with FIRRTL start with the (micro-)architecture and relies on a language based description.

9 Summary and Outlook

We presented an MDA inspired approach to automate hardware design starting at the specification level. This approach introduces three models namely The Model-of-Thing, the Model-of-Design, and the Model-of-View structuring specification, design, and views. As MDA proposes, we translate these views from the more abstract one to the next concrete one. We used for the translation Python code that is described in a way that it reflects the design and the view. Therefore, we call this Python style Template-of-Design and Template-of-View. This style is enabled by APIs generated from the Metamodel definitions of the models mentioned before.

We have proven the usefulness of the approach by generating RTL code from specifications. By doing so, we have seen the potential for productivity increase when following our proposed approach.

Next steps will introduce further transformations, especially for translating one model in an other on the same abstraction level. In addition, we want to further elaborate the best way of transformation steps being needed to translate a specification into a design. Finally, we will continuously develop new components to harden the claim and to get experience with the transformations.

Bibliography

[1] Krste Asanovi and David A. Patterson. Instruction sets should be free: The case for risc-v. Technical Report UCB/EECS-2014-146, EECS Department, University of California, Berkeley, Aug 2014.

[2] Japan Electronic Industry Development Association. *UDL-I : Unified Design Language for Integrated Circuits definition. UDL/I Language Reference Manual, Version 2.0.3, Translation from the Japanese Language Reference Manual*. JEIDA, 1993.

[3] J. Bachrach, H. Vo, B. C. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. 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*, pages 1216–1225, 2012.

[4] R. Collet and D. Pyle. McKinsey on Semiconductors: What happens when chip-design complexity outpaces development productivity. `http://www.mckinsey.com/industries/semiconductors/our-insights`, Autumn 2013.

[5] Wolfgang Ecker and Johannes Schreiner. Metamodeling. In Soonhoi Ha and Jrgen Teich, editors, *Handbook of Hardware/Software Codesign*. Springer.

[6] Wolfgang Ecker, Michael Velten, Leily Zafari, and Ajay Goyal. The metamodeling approach to system level synthesis. In Gerhard Fettweis and Wolfgang Nebel, editors, *DATE*, pages 1–2. European Design and Automation Association, 2014.

[7] J. Eker, J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[8] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[9] T. Hoshino. UDL/I version two: A new horizon of HDL standards. In *Computer Hardware Description Languages and their Applications, Proceedings of the 11th IFIP WG10.2 International Conference on Computer Hardware Description Languages and their Applications - CHDL '93, sponsored by IFIP WG10.2 and in cooperation with IEEE COMPSOC, Ottawa, Ontario, Canada, 26-28 April, 1993*, pages 437–452, 1993.

[10] IEEE. *IEEE 1685^{TM}: IP-XACT, STANDARD STRUCTURE FOR PACKAGING, INTEGRATING, AND REUSING IP WITHIN TOOL FLOWS*. IEEE.

[11] IEEE. *IEEE 1801^{TM}: Standard for Design and Verification of Low Power Integrated Circuits*. IEEE.

[12] K. Keutzer, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 19(12):1523–1543, November 2006.

[13] David Koeplinger, Christina Delimitrou, Raghu Prabhakar, Christos Kozyrakis, Yaqi Zhang, and Kunle Olukotun. Automatic generation of efficient accelerators for reconfigurable hardware. In *Proceedings of the 43rd International Symposium*

on Computer Architecture, ISCA '16, pages 115–127, Piscataway, NJ, USA, 2016. IEEE Press.

[14] Liangora Research Lab. What is MDA? Why considering BNPM.

[15] Patrick S. Li, Adam M. Izraelevitz, and Jonathan Bachrach. Specification for the firrtl language. Technical Report UCB/EECS-2016-9, EECS Department, University of California, Berkeley, Feb 2016.

[16] Stephen J Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. Model-driven architecture. In *Advances in Object-Oriented Information Systems*, pages 290–297. Springer, 2002.

[17] B. Nikolic. Simpler, more efficient design. In *ESSCIRC Conference 2015 - 41st European Solid-State Circuits Conference, Graz, Austria, September 14-18, 2015*, pages 20–25, 2015.

[18] "OMG". MDA - The Architecture of Choice for a Changing World, 2016.

[19] Johannes Schreiner, Felix Willgerodt, and Wolfgang Ecker. A new approach for generating view generators. In *Proceedings of DVCON US 2017, unpublished*, DVCON '17. IEEE Press, 2017.

[20] O. Shacham, O. Azizi, M. Wachs, S. Richardson, and M. Horowitz. Rethinking digital design: Why design must change. *IEEE Micro*, 30(6):9–24, 2010.

[21] D. Steinberg, F. Budinski, M. Paternostorno, and E. Merks, editors. *EMF Modeling Framework*. Addison Wesley, 2008.

[22] F. Truyen. The fast Guide to Model Driven Architecture.