Design Centric Modeling of Digital Hardware

(Invited Paper)

Johannes Schreiner*, Rainer Findenig†, Wolfgang Ecker*
*Infineon Technologies AG, Munich, Germany
†Infineon Technologies AG, Linz, Austria
<first name>.<last name>@infineon.com

***Abstract*—Today's dominant RTL languages, VHDL and (System)Verilog, were designed as description and simulation languages. Therefore, they have a clearly defined – but not in all cases deterministic – simulation algorithm as backbone of the language definition. Both languages have been adopted as RTL design languages but still impose a lot of simulation/synthesis mismatches. As a further disadvantage, considerable overhead can be needed to code well-known hardware patterns such as FSMs. Finally, the simulation algorithm prevents efficient simulation (e.g. two-state or cycle-based simulation) as well as advanced model analysis (e.g. X-propagation) or fosters an execution that is not in sync with the language definition. Therefore, we developed a design centric modeling approach that allows a clear specification of the design intent and provides freedom for various target HDLs and modeling styles. Since our approach is specified without underlying simulation semantics, we provide a formal definition considering only certain points in simulation traces, thus enabling various ways for simulation. To avoid syntactic sugar, we selected a metamodeling based approach, which we use as part of a model-driven generation-focused design approach.**

***Keywords*—Model Driven Architecture, Model-of-Design (MoD), Model-of-Things (MoT), Hardware Generation, Design Productivity**

I. Introduction

When RTL synthesis made its way to broad industrial application in the early nineties, over a decade of research on RTL modeling and various synthesis techniques had passed by. Neither modeling nor EDA techniques were the key to RTL synthesis success. Instead, the perception of many designers contributed to its success that only a strictly synchronous, i.e. clock related design style, helps to get a robust design in a world of increasing technological uncertainties [1]. Especially variations in propagation delay could be handled and independence of cell libraries and technologies could be achieved in this way.

Even though a *design* style was the key to success for RTL synthesis, VHDL – a *modeling* language – became the first dominant RTL language. The reason for its dominance was probably the Department of Defense (DoD) pushing it for the modeling of electronic chips and components and the DoD's funding for the development of the language. Soon, Verilog, a description language for gate level primitives became the second RTL language. Some may see this as predictable through EDA's history of competing standards such as Common Power Format (CPF) vs. Unified Power Format (UPF) or e Reuse Methodology (ERM) vs. Universal Verification Methodology (UVM). More realistically, Verilog became popular since it was widely used for gate-level simulation and a co-simulation of gate-level and RTL models was still important at that time. In turn, the benefit of VHDL over Verilog was that RTL models could be easier co-simulated with behavioral models that have been used for executable specifications and system analysis. In any case, with both Verilog and VHDL, modeling languages based on simulation algorithms were used to design digital hardware.

The consequence were specific modeling styles that tended to induce simulation-synthesis mismatches. Inferred latches, co-existing overly pessimistic and overly optimistic don't care handling, challenges with clock gating and read-before-write obstacles are only examples of the side effects of using a modeling language for synthesis.

Now, 25 years – or, in other words, 16 technology nodes – later the number of gates on a chip increased by about a factor of 10^5, yet RTL design still dominates the digital design area. In the meantime, IP reuse popped up; its effect on productivity increase is however often overestimated according to Collet et al. [2].

We see domain and application specific generation as the key for further productivity increase. Therefore, we developed a generation framework [3] to simplify building generators. Many successful utilizations in various kinds of industrial applications showed us that we are on the right track.

Initially, we used a template focused approach to directly generate HDL code. Over time, these templates became increasingly complex, not least because of inconveniences in RTL design using VHDL or Verilog. To solve this issue, we enhanced our template approach by a multi-layer generation approach with a design centric description as intermediate [3]. This approach turned out to be very powerful. One of its difficulties was that the semantic of the model items was only informally described and had neither a clear simulation semantic nor a formal definition. To overcome this, we developed a formal definition of our design centric modeling approach based on finite state machines (FSMs). These FSMs further define an execution semantic. The state trace hereof is finally used to define valid simulations. However, not only comparison of simulation traces but also synchronous assertions partially including the FSM's next state and output logic can be used to validate correct simulation.

In the remaining paper, we first introduce the design centric modeling style. Afterwards, we introduce a formal definition of the design centric modeling style. An example and a comparison with other approaches follows. The paper closes with an assessment and an outline.

II. Design Centric Models

A. Approach

The design centric RTL models we rely on, referred to as *Models-of-Design* in the following, build on the same level of abstraction as RTL modeling languages. However, they differ in the perspective they take towards the design task: instead of describing a hardware simulation along with all the semantics associated with event-driven simulation, our approach focuses on describing an image of the actual digital circuit a designer wants to construct. Each of our design centric models is an image of one instance of a certain RTL design. Every artifact in our model is thus a description of a part of the circuitry of a digital design. Examples of such artifacts are registers, multiplexers or adders and their interconnection. Metamodeling is used to classify, describe and constrain all artifacts in our models.

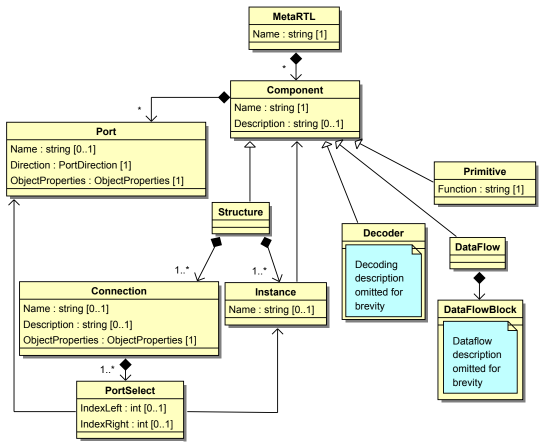


Fig. 1: Simplified Metamodel of Design Centric Models

Figure 1 shows a simplified UML class diagram of *MetaRTL*, the metamodel of our design centric models. This metamodel constrains the set of all digital hardware descriptions that are possible with our modeling approach. At the highest level, it describes an RTL design as a set of components. All these components have an interface, described by ports with associated properties (`ObjectProperties`). The components can have different realizations: they can for example be decoders, blocks describing `DataFlow` or `Primitives` such as the aforementioned registers and logic gates. In addition, components can also be realized as structure, which are different from other component realizations: structures consist of instances of other components (which can again be structures) as well as their interconnection.

B. Hardware Modeling

Based on the MetaRTL metamodel, we use a general-purpose metamodeling framework implemented in the Python programming language. This framework provides an API for entering, accessing and processing models consistent with our metamodel. Using the framework, it is also possible to import models and to run transformations and checks on them. The most important capability enabled by the framework is, however, the use of Python code for entering and generating instances of the model.

The developer can use API functions to instantiate model artifacts such as structures and to add component instances to them. In the same way, the model artifacts that describe interconnections inside these structures can be instantiated. The code a developer enters is however not the hardware description itself but the instructions to generate the hardware model when executed. In other words, it creates the hardware description as an instance of the metamodel. Thanks to powerful APIs provided by our metamodeling environment and the close fit between the hierarchical structure of our metamodel and the RTL designs we target, the code still looks very similar to what a structural HDL description would look like.

The important conceptual difference is that any conditional behavior in the code is resolved by the time the code is executed. Conditions in the generator code therefore result in conditional behavior of the model generator. If conditional behavior in the hardware is intended, this can e.g. be provided by instantiating a multiplexer.

C. Generation Flow

The large set of IP-XACT [4] applications shows that metamodeling is common in hardware design. Metamodels typically capture specification data and code generators are used to automatically generate redundant (e.g. VHDL components having the same interfaces as entity declarations) or repetitive patterns (e.g. register structures) e.g. in HDL descriptions for simulation or synthesis and in C code for firmware. Utilizing the same concept for RTL design complements those existing approaches.

The possibility to use conditionals, loops and other constructs in the Python model generators allows to automatically generate RTL models (Models-of-Design, MoD) that are tailored to a certain input configuration or specification. This input specification can in turn be provided by models of configuration and specification (Models-of-Things, MoT). In this way, it is possible to automatically generate Models-of-Design for any Models-of-Things as long as the latter adhere to a defined, known metamodel. The model generator (the Python code) is referred to as *Template-of-Design*.

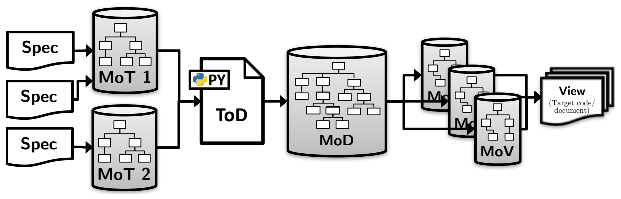


Fig. 2: Generation of target code from Models-of-Things

Figure 2 gives an overview of a code generation flow that focuses on the automated generation of Model-of-Design (MoD) instances to provide configurability and consequently increase reuse in design. The central component of our design centric RTL approach is the Template-of-Design (ToD). This template contains the information on how to build all possible design instances. In Figure 2, it is symbolized by a Python file. To produce the design instance, it pulls information from the Model-of-Things instances on its left and processes them.

Based on this information, it instantiates the Model-of-Design on the right hand side.

The Model-of-Design instance can further be processed using automated model transformations to analyze and modify the model. Such transformations could, for example, introduce the necessary connectivity and multiplexers to allow scan testing on parts of the design.

The most import use of the Model-of-Design is its transformation to target code, which can then be used e.g. by simulation and synthesis tools. There are several different approaches to generate these target views. The two more straightforward approaches rely on walking through the Model-of-Design and printing out corresponding target code for each component artifact, in each case tailored to the component type. This can either be provided by simply writing to a file with print-like statements or through the use of template engines. The latter are superior when there are significant amounts of constant target code and only small configurable parts. Providing both readable transformation code and readable target code is however a tedious task in print- and template-based approaches, requiring repetitive formatting tasks that are difficult to generalize.

For that purpose, we use an additional model that abstracts from the target view. This model, called Model-of-View (MoV) is similar to the target views' Abstract Syntax Tree. The metamodel is automatically created from an extended subset of the EBNF format that is enriched with formatting information of the target code. Based on this format, we automatically transform the AST-like representation to target code with customized, configurable formatting properties. On the front-end side, the metamodeling environment provides flexible APIs for populating the AST.

III. Formal Definition

Even though the informative character of MoD helps to escape the of RTL language simulation artifacts, its correct mapping to an HDL is vague and potentially error prone. Therefore, we present a formal semantics based on finite state machines. This allows easily defining primitives such as registers, decoders, or multiplexers as well as more complex structures such as ALUs. The composition of those FSM-based elements is also described.

The choice of FSMs to describe the semantics is based on the deep understanding that hardware and software designers as well as concept engineers usually have thereof: the engineers can choose elements for the MoD easily based on their name and validate the intended semantics instead of interpreting any given view being generated from the MoD. Additionally, this simplifies the generation and validation of additional views, for example in different output languages.

A. Formalisms

We use traditional Mealy automata to define the elements the Template-of-Design can use to populate the Model-of-Design. A Mealy automaton is the tuple of

S	is the set of states,
s_0	is the initial (reset) state,
I	is the set of possible input values,
O	is the set of possible output values,
δ	is the transition function $\delta : S \times I \to S$, and
λ	is the output function $\lambda : S \times I \to O$.

It is noteworthy that δ and λ can be arbitrary boolean functions; for example, they might very well use arithmetic operations.

Additionally, we allow a special, "stateless" case, where $S = \{\}$ and, therefore, s_0 and δ are not applicable. Furthermore, in this case, we remove S from the definition of λ and arrive at $\lambda : I \to O$. In other words, such an automaton is a pure function that maps inputs to outputs without any state.

For each primitive available to the Template-of-Design, a respective Mealy automaton is defined to describe the element's semantics. For example, a purely combinational n-bit multiplexer can be defined as a stateless automaton with $S = \{\}$, $I = I_0 \times I_1 \times I_s$ where $I_0 = \mathbb{B}^n$, $I_1 = \mathbb{B}^n$ are the inputs and $I_s = \mathbb{B}$ is the selection input. Finally, $\lambda : I \to O = I_0 \times I_1 \times I_s \to O$ where

$$\lambda((i_0, i_1, i_s)) = \begin{cases} i_0 & \text{when } i_s = 0 \\ i_1 & \text{else.} \end{cases}$$

For the simplest automaton using states, consider a D flip-flop. We can describe it using a Mealy automaton where $S = I = O = \mathbb{B}$, s_0 is defined according to the flip-flop's asynchronous set/reset, $\delta(s, i) = i$, and $\lambda(s, i) = s$. In other words, the flip-flop uses the input to define its new state and presents the current state at its output.

As a more complex example, consider a multiply-accumulate block that computes $x[n] = x[n-1] + a \cdot b$. Assume that that the bit widths for a, b and x are given as n_a, n_b, and n_x, respectively. Then, we can define a corresponding Mealy automaton[1] with

$$\begin{aligned} S &= O = \mathbb{B}^{n_x} \\ s_0 &= (0, \ldots, 0) \\ I &= I_a \times I_b = B^{n_a} \times B^{n_b} \\ \delta(s, (i_a, i_b)) &= s + i_a \cdot i_b \\ \lambda(s, (i_a, i_b)) &= s. \end{aligned}$$

B. Composition

As mentioned before, a Mealy automaton is defined for each primitive available to the ToD. The ToD then populates the MoD with those primitives and their connections. Formally, connecting primitives translates to compositing their respective automata. For an introduction to automata composition, refer, for example, to [5].

As a first, rather trivial example, an automaton M_r modelling an n-bit register can be obtained by compositing n D flip-flop automata $M_i = (S, s_{0,i}, I, O, \delta, \lambda)$ earlier in parallel. For all primitive automata, all elements except s_0 are the same. The parallel composition results in the automaton M_r where $S_r = S^n = \mathbb{B}^n = I_r = O_r$, $s_{0,r} = (s_{0,0}, s_{0,1}, \ldots, s_{0,n-1})$, $\delta_r(s, i) = i$, and $\lambda_r(s, i) = s$.

[1]For simplicity, we omit saturation or overflow handling in this example.

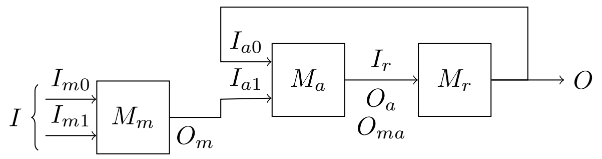


Fig. 3: Composition of a multiplying (M_m), an adding (M_a), and a register automaton (M_r) to a multiply-accumulate automaton.

As an example for a more complex composition, recall the multiply-accumulate block. While it can be defined directly as discussed before, it is also possible to composite it from three basic automata as presented in Fig. 3. First, consider two stateless automata for multiplication and addition. Let the multiplying automaton M_m be defined with

$$\begin{aligned} I_m &= I_{m0} \times I_{m1} \\ \lambda_m((i_{m0}, i_{m1})) &= i_{m0} \cdot i_{m1} \end{aligned}$$

and, similarly, the adding automaton M_a with

$$\begin{aligned} I_a &= I_{a0} \times I_{a1} \\ \lambda_a((i_{a0}, i_{a1})) &= i_{a0} + i_{a1}. \end{aligned}$$

The composition of those automata as shown in Fig. 3 yields a third stateless automaton M_{ma} where the output of M_m is connected to one of M_a's inputs:

$$\begin{aligned} I_{ma} &= I_{a0} \times (I_{m0} \times I_{m1}) \\ O_{ma} &= O_a \\ \lambda_{ma}((i_{a0}, i_{m0}, i_{m1})) &= \lambda_a((i_{a0}, \lambda_m((i_{m0}, i_{m1})))) \\ &= i_{a0} + i_{m0} \cdot i_{m1}. \end{aligned}$$

This automaton is finally composed with the register automaton M_r as described before, resulting in an automaton $M = (S, s_0, I, O, \delta, \lambda)$ where

$$\begin{aligned} S &= O = O_{ma} \\ s_0 &= s_{0,r} \\ I &= I_m \\ \delta(s, (i_{m0}, i_{m1})) &= \lambda_{ma}((s, i_{m0}, i_{m1})) \\ &= s + i_{m0} \cdot i_{m1} \\ \lambda(s, (i_{m0}, i_{m1})) &= s. \end{aligned}$$

C. Clocking schemes

In the automata introduced before, we implicitly assumed that all flip-flops are triggered by the same clock, i. e. all flip-flops will transition to their new state at the same time[2]. This allows for straight-forward modeling and analysis of synchronous circuits. Real-life designs, however, often use a number of different clock domains for different components on a single chip. Therefore, the clocks of any two automata might have one of the following relationships:

1) equal: they have the same frequency and phase,
2) generated: they have a deterministic frequency and phase relationship, or
3) asynchronous: they have no known phase relationship.

[2]As usual in RTL modeling, we do not consider clock skew or propagation delays here but leave them to the synthesis tool and timing analysis.

Automata composition as described earlier assumes that both automata share the same clock, i. e. it is only applicable in case 1.

For case 2, we can assume that either one clock is derived from a the other or, more generally, that both clocks are derived from a third one with higher frequency. For example, a 100 MHz and a 150 MHz clock might be derived from a common 300 MHz clock. In this case, we can transform both automata to share the common (higher) clock, which allows us to reduce this problem to case 1. Transforming an automaton to a use a higher clock frequency entails constructing a new automaton M' in which a "clock enable" signal is introduced as follows:

$$\begin{aligned} I' &= I \times \{en\} \\ \delta'(s, (i, en)) &= \begin{cases} \delta(s, i) & \text{when } en = 1 \\ s & \text{else.} \end{cases} \end{aligned}$$

It should be noted that clock gating, i. e. instances where a block has an input that can completely disable it, is modeled in the same way.

Turning to case 3, no such transformation is easily possible. However, RTL design often is based on the fact that blocks are locally synchronous and only use asynchronous connections between blocks. In those connections, special synchronizing circuits are used to ensure that data is synchronized correctly and no metastability can occur. For populating the MoD, we follow the same strategy: specialized synchronization primitives are provided that need to be used to synchronize data between asynchronous blocks. This results in a pattern often called "globally asynchronous, locally synchronous".

D. Ensuring the Semantics of a View

With the automata-based approach described in this section, we are able to fully capture the semantics of the resulting hardware. Still, their transformation to different Models-of-View as well as the MoV's translation to the final view are hand-written and can therefore introduce errors.

To ensure that the semantics of a view match those defined by the MoD's contents, we need to abstract certain view-specific behaviour. Most prominently, when simulating generated VHDL code, the view will usually use a number of internal processes that use signals to communicate. Due to VHDL's simulation algorithm [6], signals driven by combinational processes may carry incorrect values for one or more delta cycles.

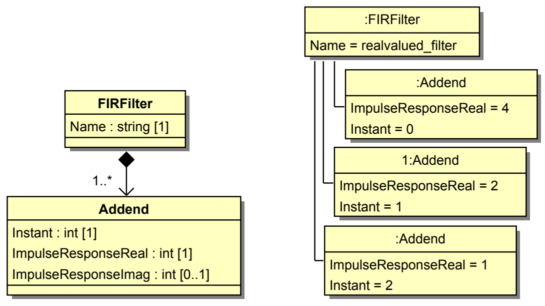
We therefore define the semantics directly on the registers and require that the interpretation of the view's register contents immediately after each clock edge must match those calculated for the MoD's contents with the same input. This approach is, for example, similar to what is used for cycle-based simulation where only the final values of output signals and register inputs are calculated [7] or assertion-based verification.

IV. Application

In this section, an example for a complete generation flow of a simple FIR filter component is provided. Based on information from a Model-of-Things about the intended characteristics of the filter, a Template-of-Design instantiates a Model-of-Design that describes the correct digital filter.

A. Model-of-Things

A suitable Model-of-Things for the task has to describe the frequency characteristics of the filter. To keep our example simple, the metamodel introduced in Figure 4a is assumed to contain the coefficients b_i from the recurrence relation $y[n] = \sum_{i=0}^{N} b_i \cdot x[n-i]$. Python and its large set of libraries for signal processing would of course also allow to derive these coefficients from a frequency-domain representation. The sample model provided in Figure 4b describes one 2nd order instance of these filters 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

```
1 fir_MoT = ... # instance of a filter specification metamodel
2 fir_filter = Structure(parent=None,
3                        in_ports=[Port('x')],
4                        out_ports=[Port('y')]))
5
6 current_delay, sum_conn = 0, None
7 current_conn = Connection(parent=fir_filter,
8                           ports=[fir_filter.ports['x'])
9 for addend in fir_MoT.getAddendsSort(sort_by='Instant'):
10     instant = addend.Instant
11     impulse_response = addend.ImpulseResponseReal
12     assert not addend.hasImpulseResponse(), \
13         'Template␣does␣not␣support␣complex-valued␣filters'
14
15     while instant < current_delay:
16         reg = Reg(parent=fir_filter)
17         current_conn.connect(reg.ports['in'])
18         current_conn = Connection(parent=fir_filter,
19                                   ports=[reg.ports['out']])
20         current_delay += 1
21
22     mul = Multiplier(parent=fir_filter,
23                      constants=[impulse_response],
24                      in_ports=[Port('in')])
25     current_conn.connect(mul.ports['in'])
26
27     instant_out = Connection(parent=fir_filter,
28                              ports=[mul.ports['out']])
29
30     if sum_conn is None:
31         sum_conn = instant_out
32     else:
33         adder = Adder(parent=fir_filter,
34                       in_ports=[sum_conn, instant_out])
35         sum_conn = Connection(parent=fir_filter,
36                               ports=[adder.ports['sum']])
37
38 sum_conn.connect(fir_filter.ports['y'])
39
40 add_clock_and_reset_everywhere(fir_filter) # recursively
41 # iterate over structure and attach clock and reset
```

Fig. 5: Template-of-Design example for generation of n-th order FIR filter

B. Generation

The listing in Figure 5 describes the part of a Template-of-Design that generates Models-of-Design for simple FIR filters with different frequency responses. To transform the Model-of-Things instance describing our 2nd order filter into a Model-of-Design, the Template-of-Design is executed.

Line 1 hints that `fir_MoT` points to the `:FIRFilter` instance from the filter specification in Figure 4. As a first task, the ToD snippet instantiates the toplevel structure `fir_filter` that will contain the entire filter once the template has been executed (line 2). Line 7 creates a connection inside the `fir_filter` and connects it to port `x` (where the incoming signal is connected to the structure).

The loop lines 9-37 iterates over all `:Addend` elements the `:FIRFilter` instance contains. It does this sorted by the `Instant` attribute. For our example filter, the loop will therefore execute three times, with the values $(0, 4)$, $(1, 2)$ and $(2, 1)$ for `instant` and `impulse_response` respectively.

The values stored in `instant` describe how many cycles the signal has to be delayed. If a filter only contains one term $y[n] = 2 \cdot x[n-10]$, the input port's signal would have to be delayed by 10 cycles before it can be fed into the multiplier. Lines 15-20 add registers for this delay to the MoD and connect them correctly. As the `instant` values are continuous in our example, this while loop is only executed once per iteration to generate our MoD.

Lines 22-28 add a multiplier which multiplies the signal on the connection stored in `current_conn` with the constant `impulse_response` factor from the MoT. In the first loop iteration for our model, `current_conn` points to the connection that is attached to the input port of the filter.

The signal on the multiplier's output port then has to be added onto the output of other previously computed addends (the terms with lower signal delays). For the first iteration of the loop, there is of course no other addend of shorter delay, which is why no adder is introduced in lines 30-36. For the first iteration, line 31 stores a reference to the connection that is attached to the output of the multiplier in `sum_conn`. The second and third loop iteration also add their multipliers to the design. In addition, they also add adders to the design. The inputs of these adders are connected to the output of the multiplier and to the connection that is pointed to by `sum_conn`. A new connection is then created and attached to the output port of the adder. `sum_conn` is set to refer to this connection. In the next iteration, it is thus attached to one of the input ports of the newly created adder.

Line 38 eventually connects the output of the last adder to the output of the `fir_filter` structure.

When the Template-of-Design is executed, it creates an instance of the MetaRTL metamodel. Figure 6 shows this instance in the representation we introduced in our formalization. It consists of basic components which have an associated FSM description and their interconnections. The composition of the basic automata provides a semantics for the overall Model-of-Design as described in III-B. From this Model-of-Design instance and its semantics, different target views can be constructed. For a `Structure` model element, our VHDL backend e.g. creates an architecture which contains components for all `Instances` part of the `Structure`. The Model-of-Design from Figure 6 would thus create one single filter entity with one input and output port. The architecture generated for this entity would then contain seven components: two registers (M_r), three multipliers (M_m) and two adders (M_a). It also contains the VHDL statement necessary to provide the connections between these components. The multipliers, registers and adders are in turn realized by their own entities. This leads to hierarchy that is identical between the target view and the Model-of-Design.

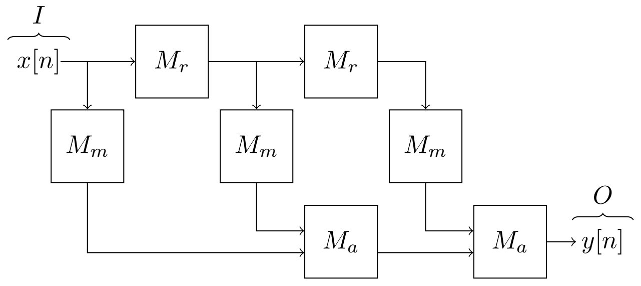


Fig. 6: Composition of multiplication (M_m), adder (M_a), and register automata (M_r) to an automaton describing the Model-of-Design generated from the Model-of-Things in Fig. 4.

C. Feasibility for larger Designs

To evaluate our approach's applicability to larger, real-life designs, we implemented a CPU core generator. The Template-of-Design of this generator can instantiate a set of CPU cores with different RISC instruction sets and a configurable pipeline microarchitecture. To create different pipeline configurations, the ToD conditionally instantiates pipeline registers and forwarding units and provides correct wiring of these components. The Model-of-Things for the CPU task captures the Instruction Set Architecture of the core in a microarchitecture-agnostic way. Our Template-of-Design uses this information to automatically derive instruction decoders and the necessary control signals for the any microarchitecture.

V. Related Approaches

As already mentioned, the leading languages for RTL descriptions – VHDL and Verilog – have simulation-based semantics. The three main related approaches therefore define an RTL synthesis semantic on top of the languages. The first to mention is the VHDL Synthesis Standard [8] (a similar version for Verilog exists too), which defines a subset of VHDL. Further, it describes the usage of the subset for describing RTL models and thus indirectly hardware. However, it leaves things like X-handling open, still tailors the execution to the simulation algorithms (thus does not leave space for optimization of simulation execution), has limited support for generation, and does not allow a hardware centric description. Second, Synopsys GTECH (see e.g. [9]) allows to compose the design from common logic elements such as gates and flipflops or more complex items such as comparators and decoders. Unfortunately, there is no way to describe their instances and connection other than using the RTL subsets mentioned above, resulting in all the problems already mentioned. Third, connectivity can be described with IP-XACT [4] which, similar to our approach, defines a metamodel for RTL connectivity. However IP-XACT does not support the specification of the behavior of the components instantiated at the leafs and has no formal basis as well.

Another hardware centric description is UPF [10]. UPF supports the definition of measures for power control, power domains and power switching in a hardware centric way. Further, due to its TCL basis, UPF is quite powerful describing the power intent in a generic way. However, UPF does not support the description of regular digital hardware and has no formal definition. Another approach worth mentioning is MetaRTL [11] (sharing its name with our Model-of-Design metamodel by coincidence), a hardware description language based on a more object oriented perspective. Contrary to what the prefix Meta suggests, MetaRTL does not define a metamodel for RTL design. Further, MetaRTL – like VHDL and Verilog – is still a modeling language and not a design language. To some extent, this introduces all disadvantages inherent in the simulation based languages as VHDL or Verilog.

A language that was designed with hardware design intention in mind is UDL/I [12], [13]. It was developed under the leadership of the Japan Electronic Industry Development Association. However UDL/I did not make it to a wide distribution. Compared to our approach, UDL/I does not have a metamodel based foundation and has weak support for generation.

Our design centric approach is motivated by the need for efficient hardware design generation. This generation aspect is conceptually also postulated e.g. by Sacham et al. [14] and Yunsup et al. [15]. To support this, J. Bachrach et al. developed Chisel [16], a language to generate and constructing hardware. The intermediate of that language called FIRRTL [17] is closest to our approach. In contrast to our approach, it describes a language and not an explicit metamodel and it has no underlying formal semantic.

Another approach that has an overlap with our metamodeling-based approach is HIFSuite's internal HIF core-language [18]. The language was developed to ease the transformation between different views such as SystemC, VHDL and Verilog. The HIF core language is therefore a superset of different hardware description languages. The largest similarity between the HIF core language and our intermediate is that they both rely on a metamodel and a hierarchical, XML-like description. In our generation stack, HIF's core language would be positioned between our Model-of-Design and our Model-of-View: It provides a higher level of abstraction than our Model-of-View as it unifies similar concepts across different hardware description languages, yet it is more concrete than our Model-of-Design as it contains the details of one or several target views.

Turning to the formal model we use for the primitives and composition, Borrione et al. define a semantic model based on hierarchical state machines for both synchronous designs written in VHDL and Verilog [19]. While their approach focuses on inter-operating VHDL and Verilog, it is applicable to other languages as well. Similarly, Kapus presents semantics based on finite state machines that extend Mealy automata with local variables [20]. Both approaches are very similar to our formalization but add complexity in describing the primitives that we tried to avoid to increase user acceptance.

In some respects, our approach is comparable to high-level synthesis (HLS) approaches: The Model-of-Things contains an high-level description of the design's intended behavior, which is automatically translated into a concrete design by the Template-of-Design. In fact, if the Model-of-Things were defined to allow arbitrary algorithms, the implementation of the Template-of-Design would need to be what is commonly understood as an HLS algorithm. However, as mentioned in the introduction, rather than using such a general Model-of-Things, we target domain and application specific generation: our approach deliberately restricts one Model-of-Things, and thereby one generation flow, to a single problem class. This, in our experience, simplifies the algorithms needed for the Template-of-Design to a manageable complexity. Real-life projects containing different problem classes will, therefore, use a variety of Models-of-Things and generation flows.

VI. Results and Discussion

In order to achieve continuous design productivity increase, we gradually widen and improve Infineon's generation approaches. One direction is the introduction of design centric modeling, since it not only overcomes challenges known from the use of VHDL or Verilog for RTL synthesis, but also simplifies the construction of generators.

Using our approach, we were able to reduce the size of generators for VHDL RTL by a factor of 10x and for Verilog RTL by a factor of 5x. While code size reduction does not necessarily translate to efficiency gains, we experienced a simplification of the design process that is underlined by these numbers. The application examples outlined in this paper illustrate the productivity increase possible thanks to the flexibility of our Python-based approach. More importantly still, Templates-of-Design permit re-usability, e.g. can create a wide range of design instances. We experienced that this generalization did not require higher effort than the implementation of point solutions.

In order to give the design centric models not only an informal semantic, we introduced a formal definition hereof in this paper. By doing this, we allow the use different simulation approaches and techniques to execute our design centric models and only request that the state traces defined by the formal model and the traces of the same states in simulation match.

As a future direction, we plan the formal definition of the Template-of-Design execution of our modeling approach in addition to the presented formal definition of the Model-of-Design created by Template-of-Design execution. Additionally, we are looking into a formal approach for checking the semantics of the generated views: the presented formalization of the Model-of-Design should allow easily deriving properties for model checking the generated views from the Model-of-Design's contents. Finally, we intend to evaluate our approach with larger designs that use multiple clocks with different synchronization methods to see whether our rather strict approach to clock-domain crossing can result in a decrease of bugs in this area.

References

[1] V. P. Robert Hum and M. G. General Manager, Deep Submicron Division, personal discussion on quantum leaps in design productivity, May. 20 2016.

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

[3] W. Ecker and J. Schreiner, "Introducing Model-of-Things (MoT) and Model-of-Design (MoD) for simpler and more efficient Hardware Generators," in *Proceedings VLSI-SoC*, 2016, p. to be published.

[4] IEEE, *IEEE 1685-2015™: IP-XACT, STANDARD STRUCTURE FOR PACKAGING, INTEGRATING, AND REUSING IP WITHIN TOOL FLOWS*. IEEE.

[5] W. Holcombe, *Algebraic Automata Theory*, ser. Cambridge Studies in Advanced Mathematics.

[6] IEEE, *IEEE 1076-2008™: IEEE Standard VHDL Language Reference Manual*. IEEE.

[7] R. Ubar, A. Morawiec, and J. Raik, "Cycle-based simulation with decision diagrams," in *Design, Automation and Test in Europe Conference and Exhibition 1999. Proceedings*. IEEE, 1999, pp. 454–458.

[8] IEEE, *IEEE 1076.6-2000™: IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis*. IEEE.

[9] *Logic Synthesis Using Synopsys, 2nd edition*.

[10] IEEE, *IEEE 1801-2015™: Standard for Design and Verification of Low Power Integrated Circuits*. IEEE.

[11] J. Zhu, "MetaRTL: raising the abstraction level of RTL design," in *DATE '01 Proceedings of the conference on Design, automation and test in Europe*, 2001, pp. 71–76.

[12] T. Hoshino, "UDL/I version two: A new horizon of HDL standards," in *Computer Hardware Description Languages and their Applications, Proceedings CHDL '93*, 1993, pp. 437–452.

[13] J. E. I. D. 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.

[14] O. Shacham, O. Azizi, M. Wachs, S. Richardson, and M. Horowitz, "Rethinking Digital Design: Why Design Must Change," *IEEE Micro*, vol. 30, no. 6, pp. 9–24, 2010.

[15] Y. Lee, A. Waterman, H. Cook, B. Zimmer, B. Keller, A. Puggelli, J. Kwak, R. Jevtic, S. Bailey, M. Blagojevic, P.-F. Chiu, R. Avizienis, B. Richards, J. Bachrach, D. Patterson, E. Alon, B. Nikolic, and K. Asanovic, "An Agile Approach to Building RISC-V Microprocessors," *IEEE Micro*, vol. 36, no. 2, pp. 8–20, 2016.

[16] 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 *Proceedings DAC '12*, 2012, pp. 1216–1225.

[17] P. S. Li, A. M. Izraelevitz, and J. Bachrach, "Specification for the firrtl language," EECS Department, University of California, Berkeley, Tech. Rep., Feb 2016.

[18] N. Bombieri, G. D. Guglielmo, L. D. Guglielmo, M. Ferrari, F. Fummi, G. Pravadelli, F. Stefanni, and A. Venturelli, "HIFSuite: Tools for HDL code conversion and manipulation," in *Proceedings HLDVT 2010*, June 2010, pp. 40–41.

[19] D. Borrione, F. Vestman, and H. Bouamama, *An approach to Verilog-VHDL interoperability for synchronous designs*. Boston, MA: Springer US, 1997, pp. 65–87.

[20] "Specification of synchronous sequential circuits using SDL and ObjectGEODE," *Computer Standards & Interfaces*, vol. 24, no. 3, pp. 257 – 274, 2002.