IP-Coding Style Variants in a Multi-layer Generator Framework

Zhao Han*†, Keerthikumara Devarajegowda*‡,
Andreas Neumeier*§, Wolfgang Ecker*†

*Infineon Technologies AG
†Technical University Munich, Germany
‡Technical University Kaiserslautern, Germany
§University of Applied Sciences Munich, Germany
Email: Firstname.Lastname@infineon.com

Abstract

With the increasing demand for domain-specific architectures, System on Chips (SoCs) need to be customized to perform the required tasks with minimal area and power consumption. However, customization brings a longer development time and more cost. Hardware generators are therefore introduced to accelerate the design process and aid Intellectual Property (IP) reuse. But the IP coding style, generated HDL files, remains unconfigurable to incorporate target design process and to achieve the desired optimization in a synthesizer.

To address these challenges, we propose to transform platform-independent design models and platform-dependent view models in our novel multi-layer generation flow. Design models capture the design intent, where deceptive components are provided to indicate hierarchies and coding styles. By fine-tuning the descriptive components, IP coding styles are configured. Furthermore, file caching and hierarchy flattening transformations are provided to further assist flexibility in the back-end. Additionally, transformations to adapt naming convention is provided as well.

To demonstrate applicability, two RISC-V SoCs are taken as examples. With generated various coding styles, the relations of IP coding styles to check-in time under version control system and synthesis time in a synthesizer is analyzed. Furthermore, efforts to develop transformations are shown.

I. Introduction

Despite rapidly developing technologies, the demand for domain-specific architectures is increasing dramatically. The term *More-than-Moore* depicts that trend. Besides, customizing a System on Chip (SoC) for different needs requires additional manual efforts, which leads to prolonged development time and increased Non-Recurring Engineering (NRE) costs. To tackle such costs, hardware generators are used to maximize the reuse of design process and codebase [1]. In a hardware generator, designers are encouraged to describe the design intent with high-level languages such as Python, Scala, etc.. Afterwards, synthesizable designs are generated according to specified configurations, the input for the generator.

Genesis2 [2] and Chisel [3] are two well-known hardware generator frameworks. Genesis2 extends SystemVerilog to enable designers to configure application-level parameters. Afterwards, human-readable Hardware Description Language (HDL) code is generated. Chisel is an embedded Domain-Specific Language (DSL) based on Scala and allows designers to utilize advance features available in modern software languages. This convenience easies the efforts to construct highly complex hardware generators. However, both of them lack flexibility in the back-end. Genesis2 is constrained by its underlying Intellectual Property (IP) implementation, while chisel can only generate one Verilog file with fixed coding style independently on the design scale.

The generated Register-Transfer Level (RTL) code must serve different use cases: simplify debugging with human-readable code, harmonize tool inconsistencies in language support, adapt coding style to different requests and follow various naming conventions. Also, the behavior shall be consistent across different EDA tools. Further, the generation flow should be flexible such that the generated code can be adapted for different aspects such as safety, power and area optimizations [4], [5]. Therefore, an important requirement of a hardware generator is the ability to generate RTL code in different target languages as well as with different coding styles. Additionally, to seamlessly integrate into different design processes, adaptable naming convention and configurable granularity of generated HDL files are required.

To address the above challenges, we have implemented a systematic approach based on a novel multi-layer generation flow [6]. The generation flow is implemented within an existing metamodel-based automation framework [7]. In the hardware generation flow, the design intent is captured in a platform-independent model, which enables the RTL code generation in different target languages such as VHDL, Verilog and SystemVerilog [8]. Further, the RTL code can be generated with different coding styles. To allow various coding styles, the design blueprint is analysed and the platform-independent model is manipulated to generate different granularity of HDL files. Fine granularity leads to better readability and file-based change analysis but prolonged synthesis time in a synthesizer

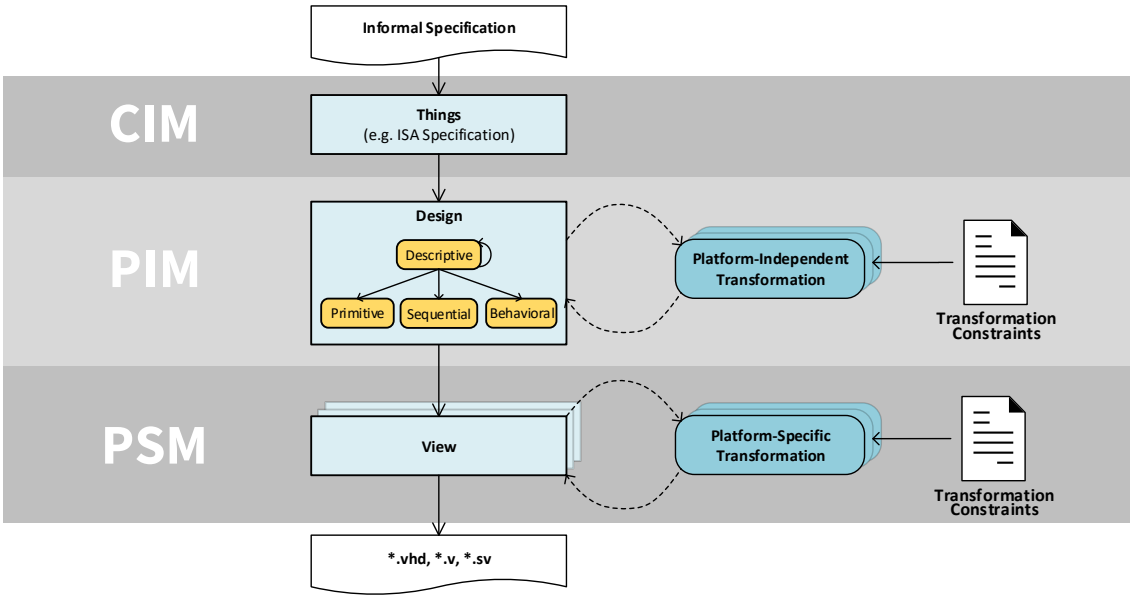


Fig. 1: Multi-layer Hardware Generator Framework

and sustained check-in in a version control system. Furthermore, by tuning the generation parameters, a suitable naming convention for a targeted design process is adjusted smoothly.

The consistent functional behavior of the generated RTL code is ensured by employing formal equivalence check between designs in different platform languages with different coding styles. Further, consistency checks such as initialization checks, FSM checks and deadlock checks are employed to ensure the quality of the generated RTL code.

To outline the paper, the novel multi-layer generation flow is described in the next section. Our approach to enabling flexibility in IP-coding styles and respective consistency checks are further elaborated in Section III and IV. Afterwards, experiments results are discussed to show the impact of our approach in Section V. In Section VI, we conclude the paper.

II. Multi-layer Hardware Generation Framework

Our hardware generation flow follows the principles of Model-Driven Architecture® (MDA®) [9] for code automation. The generation is separated into different model layers (levels) each addressing a specific aspect of the design development flow. The generation flow is outlined in Fig. 1. The uppermost model layer focuses on capturing the informal design specifications in formal specification models with well-defined semantics. Since this layer describes the design in "items", "entities" or "things", these models are called Model of Things.

The intermediate model layer is the central part of the generation flow, which is the output of the translation from formal specification models into a design model. Therefore, this model is called the Model of Design (MoD). A Python-based DSL is used to extract the specification details and describe the blueprint of the intended design [10]. In other words, the microarchitecture of the design is created in the MoD using the DSL. The design model is independent of any platform (e.g. VHDL or SystemVerilog) or technology (e.g. ASIC or FPGA). This feature of the design model enables to generate the RTL code in different platform languages, following different code guidelines and targeting different technologies.

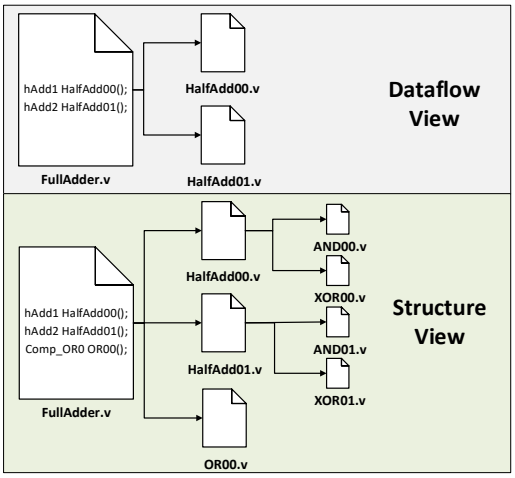
To describe the design intention, an MoD may consist of the following components:

- **Primitive Component** includes basic arithmetic and logical operators (e.g. addition, logical-and). According to platform-specific syntax, primitive components are translated to corresponding target-language representations in *View*.
- **Sequential Component** refers to components that add temporal relations between its inputs and outputs, e.g. register.
- **Behavioral Component** describes a hardware component at behavior level, e.g. FSM.
- **Descriptive Component** composes other components (as sub-components) and connections among them. A sub-component can be any kind of components, including the descriptive component itself.

Finally, in the bottom model layer, the design model is transformed into a platform-specific model shown as a Model of View in Fig. 1. The Model of View (MoV) is the least abstract model and follows the language definition of platform languages such as VHDL or Verilog [8].

```
1  class HalfAdd(Dataflow):
2      def __init__(self, *args):
3          super(HalfAdd, self).__init__(*args)
4          self.in0 = InPort()
5          self.in1 = InPort()
6          self.carry = OutPort()
7          self.sum = OutPort()
8          self.sum = self.in0 ^ self.in1
9          self.carry = self.in0 & self.in1
10 class FullAdd(Dataflow):
11     def __init__(self, *args):
12         super(FullAdd,self).__init__(*args)
13         self.in0 = InPort()
14         self.in1 = InPort()
15         self.carry_in = InPort()
16         self.carry_out = OutPort()
17         self.sum = OutPort()
18         hAdd1 = HalfAdd(self.in0, self.in1)
19         hAdd2 = HalfAdd(self.carry_in, hAdd1.sum,
       None, self.sum)
20         self.carry_out = hAdd1.carry | hAdd2.carry
```

(a) Full Adder Dataflow Description



(b) Generated HDL Files Coding Styles

Fig. 2: IP-Coding Style Variants

III. Generation of IP-Coding Style

In the generator framework, two descriptive components are available: *Structure* component and *Dataflow* component. A *Structure* component generates hierarchical views with the finest granularity of HDL files, i.e., every sub-component has a separate HDL file and is instantiated in its immediate top-level. On the other hand, A dataflow component generates compact views with coarse granularity, i.e., every sub-component is expressed directly in its immediate top-level HDL file. Descriptive components are the hierarchies in an RTL design model. By transforming the design models and view models, flexibility in coding styles are enabled (Figure 1).

A. Structurer and Dataflower

As a descriptive component defines the coding style for the current hierarchy, designers are already able to fine-tune the generated HDL files in the design intent description. Figure 2a shows the dataflow notation of a full adder, respectively only 3 HDL files are generated (Dataflow View in Figure 2b). However, by alternating to structure description, HDL files are not only generated for hierarchies but also other sub-components (Structure View in Figure 2b). However, this leads to the fixed coding style after a design description is written.

To alternate descriptive components in an MoD automatically to change the coding style in the view, transformations such as *Structurer* and *Dataflower* are provided. They iterate through the design model, locate desired descriptive components and then transform them to other descriptive components (Algorithm 1).

Algorithm 1: Algorithm for Dataflower

```
Dataflower (MoD);
Input  : Model of Design (MoD)
Output: MoD described in Dataflow
for oldComp in MoD do
    if oldComp isinstanceof Structure then
        newComp = Dataflow();
        copyPorts(oldComp, newComp);
        copySubComponents(oldComp, newComp);
        copyConnections(oldComp, newComp);
        remove oldComp;
    else
        continue
    end
end
```

B. File-Cacher

As observed from the full adder example, multiple HDL files are generated for identical functionality. For example, AND00.v and AND01.v describe the same bitwise and operation two times. Since every primitive and sequential component provides a predefined functionality, components which have the same input and output characteristics,

i.e., port size and type can be identified as the same design function. By utilizing this characteristic, *File-Cacher* transformation is enabled in which multiple similar HDL files are replaced with one HDL file. In other words, File-Cacher analyzes the design model to emit same-function sub-components only once to reduce redundancy in view. Afterwards, immediate top-levels are adapted respectively.

C. Flattener

To further reduce the number of generated files, *Flattener* flattens the hierarchies that contain an arbitrary number, given as *threshold* in transformation constraints, of sub-components. A descriptive component stores detailed information about its sub-components, e.g. number and type of sub-components. According to such information, *Flattener* iterates the MoD and locates descriptive components ought to be flattened, i.e., the number of sub-components is beneath the given threshold. During the flattening process, sub-components are moved to its new top hierarchy and connections are reworked to guarantee the design equivalence.

D. Nomenclaturer

In the design model (MoD), every component has a *Name* attribute and multiple *Port* objects. A port object has a *Name* attribute as well. With the *Nomenclaturer*, a naming convention can be smoothly adapted to different design processes. In other words, identifiers of instances and ports are changed systematically to meeting coding restrictions in the naming convention transformation.

Aforementioned transformations are classified into two categories: platform-independent and platform-dependent transformations (Figure 1). Transformations operate independently on platform-specific syntax and target technologies, such as Structurer, Dataflower and Flattener, belong to the first category. In the second category, Nomenclaturer adapts naming convention according to target coding guidelines. To formally capture required information for transformations, e.g. threshold for Flattener, naming convention for Nomenclaturer, *Transformation Constraints* is employed.

IV. Consistency Check

In the previous section, the flexibility of the generation framework to perform transformations on the design model such that the RTL code is generated in different styles is elaborated. Although the designs are generated in different flavors with the possibility to encode various nomenclature schemes, the functionality of the design must be kept intact. Towards this end, a validation flow is employed to ensure that the transformation on the design model does not alter the functional correctness and various coding guidelines required for the RTL code.

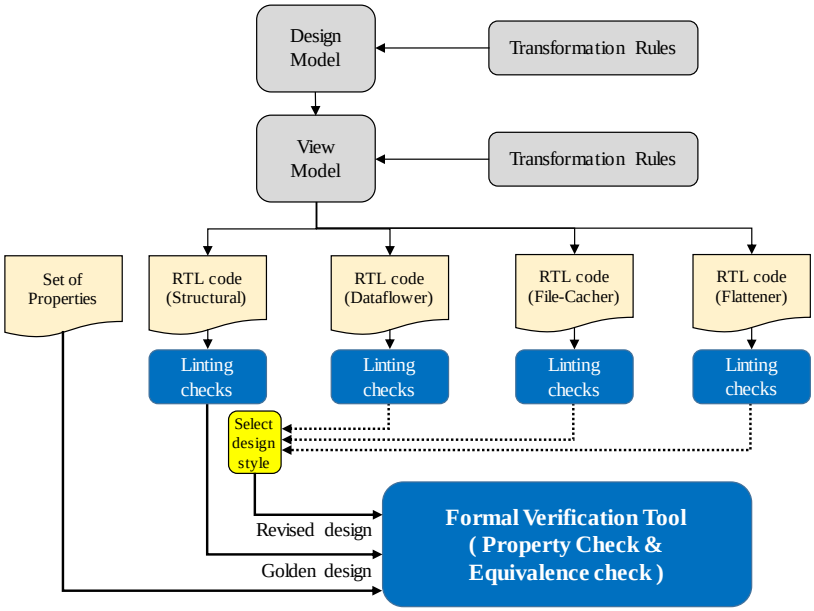


Fig. 3: Validation generation flow and verification of generated RTL code

The validation method is illustrated in Fig. 3. The design model and the view model are transformed based on the set of corresponding transformation rules as elaborated in the previous section. First, a set of linting checks are performed on the generated RTL code of various flavors. The linting includes checks for flip-flop initialization, dead-code, toggling, feedback loops, standard coding guidelines, clock domain crossing, reset domain crossing, etc.. The generation flow is built such that the generated RTL code passes the required lint checks. Additionally,

an important benefit of the generation flow is that any change required to meet the coding guidelines is performed only once in the generator code.

The generated RTL designs are verified by following the standard verification techniques including both simulation and formal verification. Towards this end, a set of properties required to formally verify a design implementation is generated following the approach proposed in [11]. The set of properties ran on one of the design versions (e.g. structural RTL code of the design). Once the design implementation satisfies the design specifications captured in the properties, an equivalence check is performed between different RTL code styles. That is, the design version that passes the property check is considered as the "golden design" model. Other design flavors are considered as "revised design" models. Running equivalence check between different RTL flavors ensures that the design functionalities are not affected by the transformation process.

V. Results

In this section, we applied our approach to generating different coding styles for two industrial RISC-V SoCs. For different generated coding styles, check-in time with Rational ClearCase® and synthesis time with Vivado® are shown. Furthermore, the development efforts of the proposed approach are shown.

A. Coding Style Variety

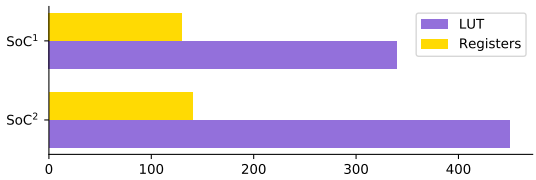


Fig. 4: Design Scale

To demonstrate the applicability, the proposed approach is applied to generate RISC-V SoCs. Each SoC is assembled with a RISC-V CPU core and several peripheral components, e.g. UART, GPIOs. Figure 4 shows the respective design resource utilization. SoC^2 differs from SoC^1 with its customisation for Machine Learning applications. To support performant computation-oriented instructions, SoC^2 is equipped with a Multiply Accumulate (MAC) unit.

TABLE I: Coding Style Variety

Design	Structural	Dataflower	File-Cacher	Flattener
SoC^1	1.3k	0.1k	0.4k	0.06k
SoC^2	12.3k	3.4k	3.8k	0.11k

Table I presents the generated HDL file number for two SoCs with different view transformations. Structurer produces always the finest granularity in view. File-Cacher performs almost the same file reduction as Dataflower for SoC^2, while it is 4 times worse for SoC^1. The reason is the MAC unit in SoC^2, which is implemented based on full adders. With such fine modular implementation, Dataflower can not reduce much file number since a full adder consists of half adders, which consists of only two gates. The Flattener (with threshold 5) produces always the fewest HDL files. Nomenclaturer affects only the naming convention, hence is not shown in the table.

TABLE II: Rational ClearCase® Check-In Time

Design	Structural	Dataflower	File-Cacher	Flattener
SoC^1	8min	0.8min	2.7min	0.4min
SoC^2	990.4min	45.5min	46.2min	1.4min

Table II shows the check-in time for different coding styles under Rational ClearCase® MultiVersion File System (MVFS). As shown in the table, more HDL files lead to prolonged check-in time. The check-in time increases proportionally with the file number until there are more than 12k files to be checked in.

On the other hand, the coding styles have influences on the synthesis time (Table III). With Xilinx Vivado®, it takes more time to synthesize the same design when more files are elaborated. Comparing the SoC^1 dataflow view and SoC^2 flattened view, similar synthesis time can be observed when the numbers of HDL files to be elaborated are close even the design scales are different.

TABLE III: Vivado® Synthesize Time

Design	Structural	Dataflower	File-Cacher	Flattener
SoC[1]	11.5min	5.0min	6.3min	4.9min
SoC[2]	97.1min	40.5min	43.0min	5.5min

B. Development Effort

The effort to develop the aforementioned view transformations is minimal (Table IV). Since all models of design have the same data structure (defined in metamodel of design), transformations are required one-time development effort and are applicable for all RTL models. Further, the Python-based language used to process design model in transformations is the same underlying DSL employed to construct an MoD on design layer. This simplicity in transformation development enables designers to write their customized transformations without additional learning efforts. Last but not least, the aforementioned transformations are equally applicable to generate VHDL, Verilog and SystemVerilog.

TABLE IV: Lines of Code of Transformations

Structural	Dataflower	File-Cacher	Flattener	Nomenclaturer
86	86	419	174	26

VI. Conclusion

In this paper, we introduce model transformations on platform-independent design models and platform-dependent view models to enable flexibility in IP-coding styles in the multi-layer hardware generation flow. By alternating descriptive components in the design models, the generated HDL files are fine-tuned. To further reduce redundancy in the view, HDL files providing similar functionalities are spared with one file. Additionally, hierarchies in the design can be flattened as transformation constraints indicate to further reduce the generated files. To integrate the generated code seamlessly in various design processes, naming convention transformation is provided as well. Further, formal equivalence checks and consistency checks are employed to assure the consistent functional behavior among different coding styles. To demonstrate the applicability of our approach, two SoCs have been experimented to show the influence of different coding styles on check-in time and synthesis time. Furthermore, developments efforts are minimal to develop the aforementioned transformations.

ACKNOWLEDGEMENTS

This work has been partly developed in the SAFE4I project which is funded by the German ministry of education and research (BMBF) (reference number: 01IS17032A). Its roots go back to the project SANITAS starting October 2009, which has been partially funded by German ministry of education and research (BMBF) (reference number: 16M30888) as well.

References

[1] 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.

[2] O. Shacham, S. Galal, S. Sankaranarayanan, M. Wachs, J. Brunhaver, A. Vassiliev, M. Horowitz, A. Danowitz, W. Qadeer, and S. Richardson, "Avoiding game over: Bringing design to the next level," in *DAC Design Automation Conference 2012*, pp. 623–629, IEEE, 2012.

[3] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing Hardware in a Scala Embedded Language," in *DAC Design Automation Conference 2012*, pp. 1212–1221, IEEE, 2012.

[4] P. Garrault and B. Philofsky, "Hdl coding practices to accelerate design performance," *Xilinx White Paper*, vol. 231, pp. 1–22, 2006.

[5] T. Marconi, D. Theodoropoulos, K. Bertels, and G. Gaydadjiev, "A novel hdl coding style to reduce power consumption for reconfigurable devices," in *2010 International Conference on Field-Programmable Technology*, pp. 295–299, IEEE, 2010.

[6] W. Ecker, K. Devarajegowda, M. Werner, Z. Han, and L. Servadei, "Embedded Systems' Automation following OMG's Model Driven Architecture Vision," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1301–1306, IEEE, 2019.

[7] W. Ecker, M. Velten, L. Zafari, and A. Goyal, "The metamodeling approach to system level synthesis.," in *DATE* (G. Fettweis and W. Nebel, eds.), pp. 1–2, European Design and Automation Association, 2014.

[8] J. Schreiner, F. Willgerodt, and W. Ecker, "A new approach for generating view generators," in *Design and Verification Conference - US*, Feb 2017.

[9] A. G. Kleppe, J. Warmer, J. B. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Professional, 2003.

[10] Z. Han, K. Devarajegowda, M. Werner, and W. Ecker, "Towards a python-based one language ecosystem for embedded systems automation," in *2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*, pp. 1–7, IEEE, 2019.

[11] K. Devarajegowda and W. Ecker, "Meta-model Based Automation of Properties for Pre-Silicon Verification," in *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pp. 231–236, IEEE, 2018.