Embedded Systems' Automation following OMG's Model Driven Architecture Vision

Wolfgang Ecker[1,2], Keerthikumara Devarajegowda[1,3], Michael Werner[1,2], Zhao Han[1,2], Lorenzo Servadei[1,4]
Infineon Technologies AG[1] - TU Munich[2] - TU Kaiserslautern[3]- Johannes Kepler University Linz[4]
Email: Firstname.Lastname@infineon.com

***Abstract*—This paper presents an automated process for end-to-end embedded system design following OMG's model driven architecture (MDA) vision. It tackles a major challenge in automation: bridging the large semantic gap between the specification and the target code. The shown MDA adaption proposes an uniform and systematic way by splitting the translation process into multiple layers and introducing design platform independent and implementation independent views.**
In our adaption of MDA, we start with a formalized specification and we end with code (view) generation. The code is then compiled (software) or synthesized (hardware) and finally assembled to the embedded system design. We split the translation process in Model-of-Thing (MoT), Model-of-Design (MoD) and Model-of-View (MoV) layers. MoTs represent the formalized specification, MoDs contain the implementation architecture in a view independent way, and MoVs are implementation dependent and view dependent, i.e., specific details in target language.
MoT is translated to MoD, MoD is translated to MoV and MoV is finally used to generate views. The translation between the Models is based on templates, that reflect design and coding blueprints. The final step of the view generation is itself part of generation. The Model MoV and the unparse method are generated from a view language description.
The approach has been successfully adapted for generating digital hardware (RTL), properties for verification (SVA), and snippets of firmware that have been successfully synthesized to an FPGA.

***Index Terms*—Design Automation, Code Generation, Metamodeling, Model Driven Architectrue, Model based Design**

I. Introduction

George Box stated in [3] ”All models are wrong but some are useful” - a well known and essential statement about models. This statement remembers, that models are an abstraction from reality, i.e., models contain some details and keep other details unrepresented. Therefore, models are at least wrong concerning abstracted details; either saying nothing about abstracted details or giving in-precise information about them. Since a model does not contain all information about reality, reality (e.g. a design) cannot be automatically derived from a model alone.

Often, model based design starts from so called executable system model. These models are also wrong - according to George Box - since they do not contain all details about the target design. Further, these models rely on an underlying execution semantic. This implicitly adds functionality from a so called simulation kernel[1] to the executable model. Therefore executable specification models may be potentially even ”more wrong”.

Nevertheless, some tools offer the functionality to generate design code from executable models. It is obvious from the arguments above, that the generated code cannot provide acceptable results. Therefore, abstract model are further manually refined, i.e., sufficient information is added, until they can act as a design blue-print[2]. Generating the design from these refined models implies that parts of the execution semantic are still present. However, the result is in most cases acceptable if the target architecture matches the execution semantic of the model. Nevertheless the price to pay is additional effort in implementing the details in the refined model.

Our approach follows the OMG MDA approach and makes use of a set of models, that are formally defined and informally easy to understand. The formal definition is not complete concerning the final design therefore leaves parts undefined. Since the semantic is defined in some aspects only, the model as such is not executable, however different executable models can be generated that comply with the formal semantic.

As other aspect, the model defines properties that are easy to understand and review. This serves another important demand of a specification; the communication among stakeholders. Therefore, we call the most abstract model also a formalized specification. In contrast, executable models are less appropriate for design intend communication.

As in OMG's MDA, model refinement is a key element of our approach. Design information is added step by step. The initial model can be as abstract as appropriate and must not serve automatic translation needs. Defined and also not defined items are clearly visible, i.e., we show what is true and what is wrong. We accept the incompleteness of models and make this visible. First refinement class is *translation*, which starts with a more abstract model and follows a blueprint - or an abstract template - of the target model. By doing so, we can refine the design to any target architecture and do not rely on an execution semantic of the model or one given target architecture. The second refinement class is *transformation*. By applying a set of transformation rules, a model can be optimized. Here, source and target model are

[1]A piece of software building the execution semantic

[2]Of course the initial models may be even more concrete, but the additional effort prolongs the specification phase

on the same abstraction level and comply to the same meta-model. The transformations are of algorithmic nature and do not follow a template-based approach. Both refinement classes are configured with additional configuration directives. The final step is the generation of views, as VHDL-RTL or C-HAL. By doing so, the design gets its execution semantic. To guarantee correctness, the abstract properties mentioned above are translated into property views and are formally proven against the design view. This paper is structured as follows: First related work is summarized. Next, OMGs MDA is sketched and our concept presented. A more detailed description of our base models, specification models, design models and view models - including view language description - follows. The paper closes with the representation of a proof of concept design as well as summary and outlook.

II. Related Work

Automating the development of embedded systems is a broadly researched field. Several industrial as well as academic solutions exist for this purpose. Leading survey houses predict a huge demand for low-cost, energy efficient, feature rich Internet of Things (IoT) in the near future. As these IoT devices need to be as cost efficient as possible, allocating huge budgets for their development (NRE costs) is not a sustainable solution [17]. Different approaches for automating embedded system are proposed in [4], [5], [11].

Apart from the long history of Hardware Description Language (HDL), such as VHDL and Verilog, languages like SystemC are proposed in order to bring the hardware design and simulation to system level. Towards more efficient hardware design, automated generation of Register Transfer Level (RTL) has received a lot of attention both in academia and industry.

Learning from modern software engineering, Chisel [1] is proposed, which enable high level programming language with hardware description capability. Different hardware designs can be automatically generated due to parameterizable hardware generator. FIRRTL [15] is the intermediate language empowering Chisel with generic description of hardware design, afterwards executable codes in Verilog can be generated to construct circuits. Comparing to our approach, it loses the link to specification and has no underlying formal semantic.

The approach in [13] generates hardware accelerators from the description in ANSI/ISO standard C. But it limits itself, i.e., it works only with a deterministic soft CPU core, its tool chain and its memory system, therefore the applicability of this approach on different SoCs are problematic.

Functional verification of the hardware designs is an important aspect of the embedded system development process as well. In practice, verifying functional correctness requires more than 50% of the hardware development cycle. For the functional verification of hardware designs, several automation techniques that automate the assertions (properties) exist [6], [18], [21]. To the best of our knowledge, none of the techniques follow MDA vision for code generation.

Approaches of [2], [14], [23] adopt MDA/UML in the co-design methodology in order to master the challenges of faster product development and steadily growing silicon complexity. In [12] a refined MDA is proposed, which masters the issues arising in embedded systems. Their approach is tailored to handle concurrent operations and interaction of software components with strict timing using a meta-programmable Generic Modeling Environment.

III. Approach

A. OMG's Model Driven Architecture (MDA)

The Object Management Group (OMG) proposed the idea of MDA as an approach for using models in software development [16], [22]. The MDA principle was proposed to help reducing complexity, lower costs and fostering re-usability. A "model" has well-defined semantics and is an abstracted version of the intended system. There are three main models involved in MDA, namely ***C****omputation* ***I****ndependent* ***M****odel* (CIM), ***P****latform* ***I****ndependent* ***M****odel* (PIM) and ***P****latform* ***S****pecific* ***M****odel* (PSM). These models are involved in model-to-model transformations in which the resulting model is a less abstracted model of the intended system.

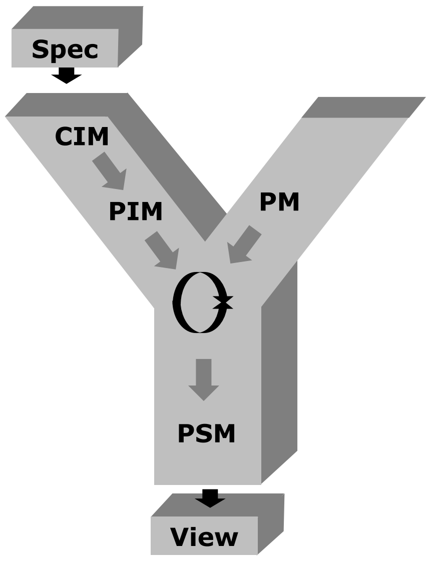

Fig. 1: Model Driven Architecture as a Y-Chart

Fig. 1 shows different models involved in MDA and their dependencies in a Y-Chart. The left branch of the Y-Chart describes the platform independent details of the system, while the right branch adds the platform specific details. The central pillar combines both the details to implement the intended system in a required platform.

The development cycle for building code generators starts from requirements/specifications of the system as shown in fig. 1. Generally, the requirements are available in an informal language. These requirements are captured in a formal model called CIM. CIM focuses on the high-level details of the system without considering the structure and processing of the intended system.

PIM is the resulting model of the model transformation of CIM. Platform independent model describes the structure and processing of the system without considering the semantics of the platform languages. Platform independence enables PIM to be mapped into multiple platforms, guided by the ***Platform Model*** (PM).

B. MDA Adaption to Embedded System's Automation

A basic embedded system is comprised of a processing unit, one to many peripheral devices, interface units, memory units and an embedded software program to realize a specific application. That is, the hardware and the firmware are the two main building blocks of an embedded system [8]. We follow the principles of MDA for hardware and embedded software development. In addition, to guarantee the functional correctness of the generated hardware, properties for hardware verification are also automated following the MDA approach.

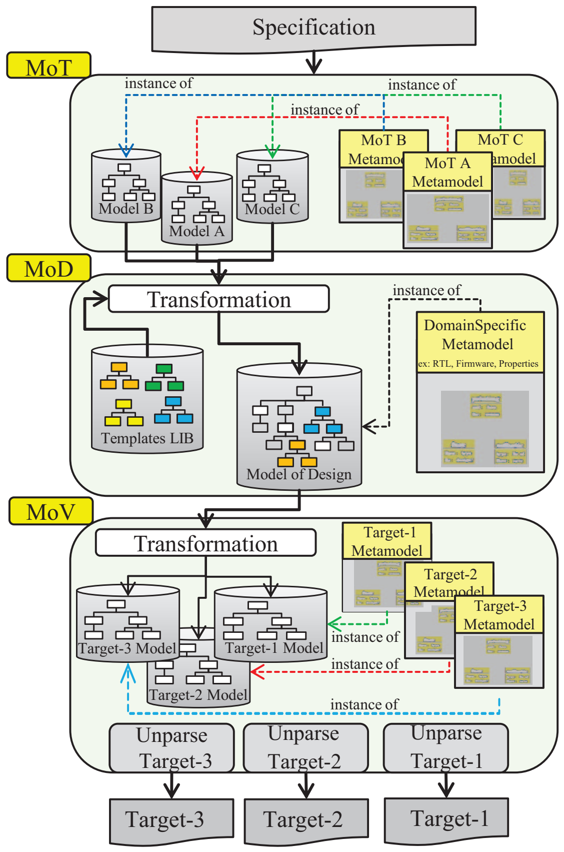

Fig. 2: Model Driven Architecture applied to embedded systems' automation

As mentioned earlier, Infineon uses a metamodel based automation framework for code generation [9], [10]. In our adaption of MDA, we make use of the automation framework to efficiently develop the code generators. A generic depiction of our adaption of MDA is shown in fig. 2.

The specification and requirements of an embedded system are generally captured in an informal language. The availability of such requirements in a formal language/format with clear semantics is rare. Hence, the first step in our flow is to translate these informal specifications to formal specification with clear semantics. We call our formal specification models as Model-of-Things (MoTs). MoTs correspond to CIM in fig. 1. Only the high level requirements of a system are captured in these models.

The high level requirements in MoTs are transformed in Model-of-Design (MoD). MoD corresponds to PIM in fig. 1. The transformations are coded in Python and define a domain specific language (DSL) of the respective domain (RTL, Properties, Embedded software). A domain specific metamodel is developed that defines the abstract structure of a system. MoD is an instance of such a domain metamodel. For such a metamodel, the underlying automation framework creates an infrastructure that aids the description of MoD through Python coded DSL. For example, the metamodel of RTL (VI-A) defines the components, their attributes and abstract relationships among components. An instance of this RTL metamodel, i.e., MoD, defines the hardware architecture of the system. Transformations are developed with high degree of configurability and hence, while building a new transformation, the existing transformations are re-used as shown in fig. 2.

Finally, the MoD is transformed in Model-of-View (MoV). MoV corresponds to PSM in original MDA description and the platform model (PM) shown in fig. 1 corresponds to target specific models (ex: for VHDL, grammar notation (EBNF) of the language represents the PM). These platform specific models contain specific implementation details of the platform. Since MoD is platform independent, it can be mapped to any platform specific model to realize the view files in a specific platform. Based on the requirement of the target languages, "*target metamodels*" and corresponding "*unparsers*" are developed to complete the generation flow.

C. MDA Application to Embedded System's Architecture

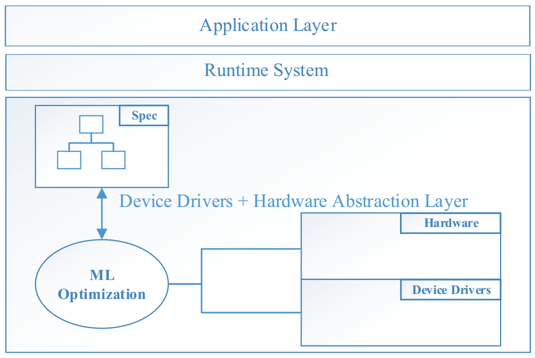

Fig. 3: Machine Learning for the optimization of embedded systems' automation

The MDA approach allows several degrees of freedom within the configuration of the embedded system, with respect to the final output of the generation flow. An example of that is the selection of specifications able to satisfy Hardware and

Firmware requirements, e.g. a certain maximal area, power consumption or performance.

This gives the possibility to the Hardware Designer using the MDA pattern to generate code not only in an easy and automated fashion, but also in an optimal way. Fig. 3 shows how an ML driven optimization can be used, jointly to an MDA architecture, for choosing an optimal specification with respect to the requirements. Here in fact, the ML optimization algorithm is able to evaluate the output of the specification towards one or more objectives (e.g. area, performance, etc) and change the configuration parameters of the specifications. This leads to the optimal compliance to the given objective or set of objectives (Pareto front).

IV. Base Models

Using an in-house metamodel based code generation framework [10], an infrastructure is generated in conformance to a metamodel. The customizable framework provides model creation, model merging, model comparator and many more required features to build the generators in an efficient, maintainable manner. Base models, i.e., Type Model, Operator Model and Expression Model, are designed to provide a generic intermediate description. They are intended to be used in different layers of our approach, which avoids the effort to develop redundant metamodels and improve the overall code quality and re-usability. Self-evidently, metamodels reference base models only on request and through model merging. Model transformation is easier to implement because of the interoperability across various metamodels. Following MDA, our approach makes it easy to reuse metamodels due to the structural refinement and maintains the consistency with automatic model transformation.

A. *Type Model*

The Type Model is intended to use underlying semantics of hardware interconnects to describe data types and interpretations. Therefore, it supports the specification of any kind of bundles of wires together with some optional hardware properties (e.g., signed interpretation and Endianness). The name of types improves readability of the generated targets and simplifies referencing types in specifications. Type compatibility is however independent on type names or other hardware properties. Instead, type consistency is only dependent on the size, i.e., the bitwidth or hardware wires, of any two types.

B. *Operator Model*

The Operator Model defines a set of supported operators within the framework, both classical operators from software program and specific hardware operators (e.g., add-with-overflow and select). For each operator, the size of the output depends on the size of the inputs. Relations between the inputs are specified. There is however no size limit as such. Operator nodes can be connected to binary or numeric literals in any combination or referenced to objects with properties defined by the Type Model.

C. *Expression Model*

The expression model is a special case of a dataflow model. The expression model uses operator model to define an expression tree with an executable semantics. The dataflow model is a directed graph with the same operators as the expression model and in addition operators with more than one output. Further, the dataflow model can include a delay operator as node.

V. Specification Models

Model of Things (MoTs) are structured specifications which express properties for a specific component or set of components, i.e., subsystems. Notation of expressions and types can be done using the Base Models.

They represent the design intent in an easy readable and structured way. Therefore, the MoT can be used as both, a communication medium and a source for design generation. It is important to mention that the MoT does not include any implementation specific details or any constraint concerning possible implementations.

The MoT is used as starting point for generation of both design and verification as well as for both, hardware and software.

VI. Design Models

Model of Designs (MoDs) consists of information about ”how” an design is implemented. The MoD is created with an implementation specific but platform independent template of design (ToD), which specifies a blue print of the target architecture (see fig. 2). If properties are involved, a Model of Properties (MoP) and a Template of Properties (ToP) is used instead.

Therefore, the templates ToD respectively ToP fall in the category of translation. The templates extract information form the MoT in order to create the design/property model. The creation process can be further parameterized with so called configurations.

As designs, both MoD and ToD can be hierarchical as well. Since, embedded MoDs can have their own MoT as well, a second task of MoT is to create MoTs of embedded components. MetaCSC (see sec. VI-B) is a good reprsentative following this strategy.

A. *MetaRTL*

MetaRTL describes RTL in a high abstraction level, it is an DSM consisting of one root node and components with realization alternatives: Structure, Primitive, Dataflow as well as Finite State Machine (FSM). Structure realization implies structural description, where the designer specifies every connection among hardware components. Dataflow realization provide additional dataflow notation on top of Structure simplifying annotation. Besides, FSM is realized with another metamodel describing the structure of FSM. Also all design models make use of the Type and Expression Model. Depending on the model, the Expression Model is used to generate structural primitives, dataflow nodes, or expression nodes.

To simplify the template description, all expressions and derived notation elements can have any kind of inputs. The model notations adopt accordingly. There are also other goodies available as automated clock and reset connection and object size as well as automated object name to port name propagation. Further details about the automation mechanisms are described in [19].

With diverse hardware specific operators from metaRTL, configurable components are constructed and contribute themselves again into metaRTL framework. With this iteration process, hardware designer have a rich library full of basic components and customizable templates (creating complex components).

B. *MetaCSC*

A common metamodel describing the register interface used in several hardware peripherals is provided by the metaCSC. It is mainly structured in a set of registers, a set of bit fields, and a mapping between both. Therefore, bitfields can be specified independently from registers and can be remapped as appropriate. Each bit field individually specifies read and write access conditions. Furthermore, the interface design is kept very generic and adjustable to any peripheral. This enables the designer to adapt architectural properties like width of the address bus or the size and endianness of all registers to the defined application. In the proof-of-concept (see sec. VIII), this description is not only utilized to generate a custom RTL architecture of the interface, but serves also as a source to generate the HAL.

C. *MetaProp*

For automating the properties for design verification, a metamodel is developed to describe the high-level expression tree of a temporal language. MetaProp uses Expression Model described earlier for enabling expression definition. The metaProp metamodel is also extended with so called system tasks to simplify the MoP definition via Python coded DSL. An instance of this metamodel captures a specific behavior expected from the Model-of-Design and is called Model-of-Property (MoP). Although the MoP is not a design model, the underlying principle is similar to the MoD. The hardware behavior can be modeled as boolean predicates over different time-points. The MoP is a platform independent way of describing temporal traces. To reason about the hardware behavior, we define the LTL semantics for MoP [7]. Similar to metaRTL, the MoP is mapped to platform specific models to generate the properties in different target languages. The property generation flow at present, supports properties in SystemVerilog Assertions (SVA) and InTerval Language (ITL).

VII. View Models

Based on the preferences of the designer and type of the design model, different languages are used during the development phase of a product. MetaRTL is basically used to generate HDL code, while MetaCSC is also considering C as a target view layer and MetaProp is targeting verification languages like SystemVerilog.

The Model-of-View (MoV) is the last layer of the code generator process and therefore also the layer closest to the target builds - the previously mentioned views. The core framework of this layer is the View Generator, which is turning an EBNF alike notation, so called View Language Description (VLD), into a metamodel and its API. Beside those, also an unparser is created from the VLD, which provides an automatism to convert an AST-like model into the final view in a formatted way.

In general, a VLD consist of a series of syntax rules. A rule to generate a "Switch" statement is shown in listing 1. The rule itself is composed of terminals (static strings or attributes), non-terminals (production rule) and unparsing hints (e.g., formatting). A detailed explanation regarding the VLD syntax and the implementation is given in [20].

```
Switch = 'switch(', Expression, ')\n',
         $indent()$(+SwitchItem, '\n'+);
         $indent()$([DefaultItem, '\n']), '}';
```

(a) C - VLD

```
Switch =[<Label>, ' : '], 'CASE ', Expression, ' IS\n',
         $indent()$(+CaseItem '\n'+),
         $indent()$([DefaultItem '\n']), 'END CASE';
```

(b) VHDL - VLD

```
Switch = 'case (', Expression, ')\n',
         $indent()$(+CaseItem, '\n'+),
         $indent()$([DefaultItem, '\n']), 'endcase';
```

(c) Verilog - VLD

Listing 1: Simplified snippet from View Language Description for different target languages describing a Switch statement.

As a summary, it may be stated that different VLDs can describe different formal languages, resulting in different metamodels or coding styles. Moreover, by defining the syntax and formatting of a language, we set constraints to the MoVs and guarantee syntactical correctness. This reduces debugging effort significantly.

Currently we use the approach to generate the HAL in C from the register interfaces defined in the MetaCSC. Additionally, hardware related formats as VHDL and Verilog are generated by transforming the architectural elements defined in the MoD into MoV.

VIII. Proof of Concept

Using our metamodeling toolkit, enables to auto-generate embedded components by extracting information from a given specification model. This hardware development flow is used to generate configurable interrupt controller, register interfaces, timer and many other common components. In order to prove the concept, we specified a design consisting of an autogenerated RISC-V system extended by a DMX and UART interface.

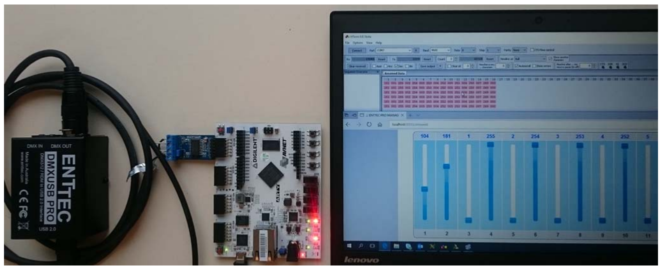

Fig. 4: DMX Receiver test-setup using an USB-to-DMX dongle

The validation of the developed design for a real application is done on the Digilent Arty A7 board. Figure 4 shows the test setup. We generate an adjustable DMX data by the Enttec software. The USB-to-DMX dongle converts the transmitted DMX data into a valid data-stream, which is fulfilling the DMX standard requirements. The DMX receiver on the FPGA is configured to extract all data slots. Subsequently, the received and filtered data are transferred back to the PC via the Uart transmitter. The comparison between transmitted and received data confirms the correctness of the generated design. Moreover, we successfully verified the design and increased the test coverage through behavioural simulations.

IX. Summary and Outlook

This work describes the python based MDA framework developed by Infineon that supports constructing advanced hardware designs. The used approach defines abstraction layers with metamodels. Furthermore, our approach is not only covering the hardware construction, but also the verification and partly the generation of embedded software. We successfully prove the correct application of our concepts on a demonstrator that is synthesized to an FPGA.

Currently, development of a firmware generator is ongoing and will be published on its own. Towards functionality and usability of the generated APIs from metamodels, extensions for API are programmed by hand. Since there are redundancy and similarity comparing extensions among API extensions of different design models, a generic methodology focusing on automatically generated API extensions is invested.

X. Acknowledgements

This work has been partly developed in the ITEA3 project COMPACT (reference number: 16018) and is partially funded by the German ministry of education and research (BMBF) (reference number: 01IS17028). 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] 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.

[2] Lossan Bondé, Cédric Dumoulin, and Jean-Luc Dekeyser. Metamodels and mda transformations for embedded systems. In *Advances in design and specification languages for SoCs*, pages 89–105. Springer, 2005.

[3] G. E. P. Box. Robustness in the strategy of scientific model building. In *Robustness in Statistics*, page 201–236, 2005.

[4] Robert Brzoza-Woch, Łukasz Gurdek, and Tomasz Szydlo. Rapid embedded systems prototyping-an effective approach to embedded systems development. In *2018 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 629–636. IEEE, 2018.

[5] Chris Conger, Ross Hymel, Mike Rewak, Alan D George, and Herman Lam. Fpga design framework for dynamic partial reconfiguration. In *Proceedings of Reconfigurable Architectures Workshop (RAW)*, 2008.

[6] A. Danese, T. Ghasempouri, and G. Pravadelli. Automatic extraction of assertions from execution traces of behavioural models. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 67–72.

[7] K. Devarajegowda, J. Schreiner, R. Findenig, and W. Ecker. Python based Framework for HDSLs with an underlying Formal Semantics. In *Proceedings of the 36th International Conference on Computer-Aided Design*, ICCAD '17, New York, NY, USA, 2017. ACM.

[8] Wolfgang Ecker, Volkan Esen, Thomas Steininger, and Michael Velten. *HW/SW Interface*, pages 95–149. Springer Netherlands, Dordrecht, 2009.

[9] Wolfgang Ecker, Michael Velten, Leily Zafari, and Ajay Goyal. Metamodeling and code generation -the infineon approach. In Wolfgang Mueller and Wolfgang Ecker, editors, *MeCoES - Metamodelling and Code Generation for Embedded Systems: Workshop with ESWEEK.*

[10] 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.

[11] James Grenning. Agile embedded software development. *ESC Boston*, 2011.

[12] Gabor Karsai, Sandeep Neema, and David Sharp. Model-driven architecture for embedded software: A synopsis and an example. *Science of Computer Programming*, 73(1):26–38, 2008.

[13] David Lau, Orion Pritchard, and Philippe Molson. Automated generation of hardware accelerators with direct memory access from ansi/iso standard c functions. In *null*, pages 45–56. IEEE, 2006.

[14] S. Lecomtea, S. Guillouard, C. Moyb, P. Leray, and P. Soulard. A co-design methodology based on model driven architecture for real time embedded systems. *Mathematical and Computer Modeling*, 53(3-4):471–484, 2011.

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

[17] Maria Dolores Valdes Pena, Juan J Rodriguez-Andina, and Milos Manic. The internet of things: The role of reconfigurable platforms. *IEEE Industrial Electronics Magazine*, 11(3):6–19, 2017.

[18] Frank Rogin, Thomas Klotz, Görschwin Fey, Rolf Drechsler, and Steffen Rülke. Automatic generation of complex properties for hardware designs. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, pages 545–548, New York, NY, USA, 2008. ACM.

[19] J. Schreiner, R. Findenig, and W. Ecker. Design Centric Modeling of Digital Hardware. In *IEEE International High Level Design Validation and Test Workshop, HLDVT 2016*, pages 46–52, 2016.

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

[21] D. Sheridan, L. Liu, H. Kim, and S. Vasudevan. A coverage guided mining approach for automatic generation of succinct assertions. In *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*, pages 68–73, Jan 2014.

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

[23] Jorgiano Vidal, Florent De Lamotte, Guy Gogniat, Philippe Soulard, and Jean-Philippe Diguet. A co-design approach for embedded system modeling and code generation with uml and marte. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 226–231. European Design and Automation Association, 2009.