A New Approach for Generating View Generators

Johannes Schreiner, Felix Willgerodt, Wolfgang Ecker
Infineon Technologies AG, Munich, Germany
Technische Universität München, Munich, Germany
Email: <first name>.<last name>@infineon.com / <first name>.<last name>@tum.de

***Abstract*-Code generation is an important key to further productivity increases in IC design. It also helps to bridge design gaps and to guarantee consistency across heterogeneous systems. Here, different views such as VHDL and C have to be automatically generated from input data formats (e.g. from IP-XACT metadata). We propose a novel approach to an essential task of code generation tools: the assembly of the target view. Our framework consists of an intermediate similar to Abstract-Syntax-Trees, an API to construct this intermediate and an automatism to generate the final view. All these components are automatically generated from a single source: the View Language Description. Our framework makes developing generators significantly easier while increasing their maintainability and the quality of the generated code. In addition to reducing generator code size by over 50%, our approach significantly reduces the number of debug cycles during generator development and increases generator readability.**

I. Introduction

The growing complexity of ASIC products over the last decades was accompanied by a steady increase in non-recurring engineering costs for these designs. Large teams of developers are needed to derive and implement individual designs, tailored to certain application scenarios. Expert designers use their knowledge to narrow down the vast solution space to a certain point solution. This is an iterative process that is repeated from scratch for every new design and gets more difficult with increasing design complexity. Code generation is named as a promising approach to capture and automate this tedious, manual process [1], [2], [3]. Instead of providing a point-solution for one application scenario, code generation approaches try to map the process: they provide reusable generators that can be utilized in a wide set of designs without manually reiterating the design process. An integral part of code generation approaches is the *view generation* for views such as HDL code, C code, test vectors or documentation which are then passed to downstream tools.

This paper first describes the state of the art in view generation and then illustrates the issues that accompany it. We then give an overview of the existing *Metamodel-based* approaches used at Infineon to structure and improve code generation. Our new approach to the generation of view generators is part of a multilevel generation approach inspired by the *Model-driven Architecture* principle which is introduced in Section III. We then describe the main contribution of this paper: our approach to *generating view generators*, consisting of the so called *Model-of-View*, its Metamodel and the mechanism to generate target views from these models. All these components are automatically provided using a single description of the target views language, the *View Language Description (VLD)*. After introducing the basic concept, several caveats and extensions are detailed that give our generation approach the full expressiveness necessary for any meaningful target view, effectively allowing it to replace any existing view generation approaches.

II. View Generation State of the Art

The naïve and commonly used approach to view generation is outlined in this section. In code generation systems, the view generator contains code that traverses the data structures internal to the view generator. To generate the actual content of the target view, the generators typically use print-like statements which output view code into files. This simplistic approach can be structured and improved by the use of template engines [4]. Template engines originate from web development, where they are used to insert dynamically generated content into large amounts of constant view content. They are superior to print-like view generation particularly when the amount of configurability in target views is low compared to the overall view size.

While this approach has shown to be very flexible and useful in our daily work, we identified a set of problems which are collectively a major obstacle to higher productivity:

- Convoluted, hard to maintain code: with increasing level of configurability, the readability of view generators suffers. This is mainly caused by the opaque mix of snippets containing static target view content and code that makes up the generator logic.
- Syntax and formatting of the generated code is not inherently correct. As a consequence, compile-and-debug iterations are necessary to generate output in the desired shape. It is further hard to write generic, re-usable code for tasks such as indentation and formatting that are necessary for a wide set of views. Further, target views are typically hierarchically structured, requiring the opening and closing of scopes when new hierarchy levels are introduced.
- Mismatch between the order of code generation and the order in which the generated artifacts appear in the target views. For many of the targeted views, it is necessary to change different positions of the file for every artifact that is introduced into the view. For example, variables, types or other names often need to be declared in one location of a file before they can be defined and used in another location. As view generators eventually need to produce coherent, continuous code, it is necessary to first run generator logic to figure out all declarations and then to reiterate the logic to provide all definitions. From a generation perspective, it is more intuitive to think in terms of generated artifacts. For an exemplary HDL view, it is e.g. desirable to once figure out all signals that need to be assigned and to then insert both the signal declarations and the assignments for the signals into the view.

III. Metamodeling for structured Code Generators

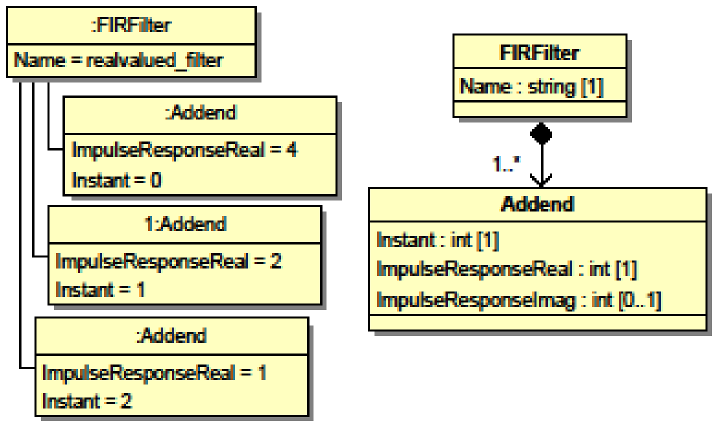
Code generation is heavily used at Infineon and plays an increasingly important role in increasing productivity across a wide range of heterogeneous products. The code generator systems are implemented in the Python language, using Mako [5] as a template engine for view generation. The flexibility and comprehensiveness of the Python ecosystem has proven to be very helpful to quickly implement powerful generators.

A. Metamodeling

A dynamically typed language such as Python is however accompanied by a tendency to a lack of structure and formalization in the data structures used for view generation. Metamodeling is a very powerful approach to guarantee interchangeability of data between different generators and to foster re-use of generator code. It also plays an important role in bridging design gaps across heterogeneous systems, where different target views have to be automatically generated from common sources to guarantee consistency. The success of the metadata interchange format IP-XACT which supports packing, integrating and re-use of IP components is a good example for the necessity of such well-defined, common sources of metadata [6]. Instead of focusing on a certain metadata interchange format, the remainder of this section describes the metamodeling idea in general and shows how metamodeling environments facilitate working with models and their Metamodels.

In Metamodeling, every model is formalized by a so called Metamodel. These Metamodels define structure, constraints and other properties of models. Figure 1a and 1b provide an example of a model and its metamodel, visualized through a UML instance diagram and the associated UML class diagram. The Metamodel in Figure 1b describes the recurrence relations of digital filters of the shape $y[n] = \sum_{i=0}^{N} b_i x[n-i]$. According to the metamodel, every coefficient is a complex-valued integer, represented through its real and imaginary components. Figure 1a contains one legal instance of this metamodel, describing the filter instance with the recurrence relation $y[n] = 4 \cdot x[n] + 2 \cdot x[n-1] + 1 \cdot x[n-2]$.

The purpose of a metamodeling environment is to provide the interface to access, create and transform model instances utilizing the description provided by their Metamodels. An auto-generated API is a key component provided by the metamodeling environment. Figure 1c sketches a very simplistic API for the Metamodel in Figure 1b. While common metamodeling environments provide more powerful APIs, this covers the basic concept of API generation within metamodeling environments: the API provides access to all objects and their properties of every model of the metamodel. The model data can usually be accessed and modified with getter and setter methods respectively.



```python
class FIRFilter:
   def getName()
   def setName()
   def addAddend(Instant,
                 ImpRespReal,
                 ImpRespImag)
   def delAddend(pos)
   def getAddends()

class Addend:
   def getInstant()
   def setInstant()
   def getImpulseRespReal()
   def setImpulseRespReal()
   def getImpulseRespImag()
   def setImpulseRespImag()
```

(a) Model (b) Metamodel (c) API Example

Figure 1. Example of a Model, its Metamodel and a sample of a simplified Python API that is generated thereof

For a more detailed introduction to metamodeling and typical workflows, see literature (e.g. [9]) on the Eclipse Modeling Framework (EMF), an open source metamodeling framework embedded in the Java and Eclipse ecosystems that is – in terms of functionality – similar to Infineon's proprietary framework.

Our multilevel generation approach and the view generators part of it are implemented based on Infineon's framework. It is however possible to implement similar functionality independent of a particular metamodeling environment. EMF is for example well-suited to implement the view generation approach we introduce in this paper.

In addition to the auto-generated APIs, metamodeling environments provide a wide range of tools. Common functionalities include the tools necessary to automatically read and write models to and from storages such as XML files, spreadsheets and databases and assistance in visualization and entry of models.

Metamodeling is already successfully used in combination with naïve view generation approaches. Here, structured data source (e.g. an XML file, a relational database or a spreadsheet) is read into the API provided by the metamodeling environment. Depending on the structure of the data source, the tools necessary for this step can be automatically provided by the metamodeling environment. When data is read into the API, a view generator is used to access the API and to dump view code. These view generators are manually implemented using the methods we described in Section II and are accompanied by the problems we listed.

B. Model-driven Architecture

We already mentioned that our view generation approach is part of a multilevel code generation system inspired by the Model-driven Architecture (MDA) idea. Model-driven Architecture is a vision of the Object Model Group (OMG) for automated code generation. Instead of using one simple model and its Metamodel as sketched in Section I.A., MDA relies on a series of model instances, each instance of different Metamodels. Starting with an abstract model that is close to the specification of a system, a series of automated refining transformations is used to derive models closer to the implementation. The most concrete of these models is used to generate target views.

In our adaption of the MDA idea to hardware design, we use three modeling layers, the *Model-of-Things (MoT)*, the *Model-of-Design (MoD)*, and the *Model-of-View (MoV)*. Figure 2 illustrates how an abstract specification is read into several Models-of-View, the arrows show how the data stored in the models is transformed and transferred to other more precise models until a target view can be derived.

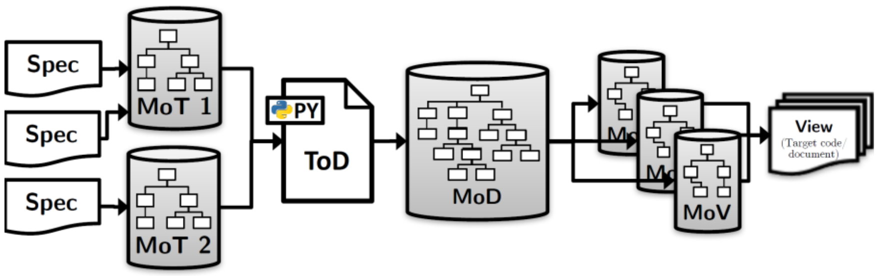


Figure 2. Three modeling layers of our MDA approach and their transformations

The Model-of-Things (MoT) is the most abstract of the three modeling layers. It is designed to capture data from requirements and the specification. The Metamodel of the MoT depends on the concrete design task at hand. A suitable Metamodel for the design of a CPU core would for example be shaped to capture the instruction set of CPUs, while a MoT for a digital filter might be shaped to describe transfer characteristics of the filter from a specification perspective.

The Model-of-Design (MoD) describes the microarchitecture of a system on RTL. The MoD is the core component of our methodology. For every generator based on our generation system, the Model-of-Design adheres to the same unique Metamodel: *MetaRTL*. MetaRTL models provide a structural view on the design's microarchitecture on RT level. Unlike models in RTL languages such as Verilog or VHDL, our models take a design centric view and avoids the introduction of simulation semantics. An instance of MetaRTL thus provides a view on the components of a design, their hierarchy and their interconnection.

The Model-of-View (MoV) is the model which is closest to the target views produced by the generation system. The Model-of-View has a Metamodel tailored to the language of the target view. The approach to view generation in this paper provides an automated way to derive the Metamodels for a Model-of-View, an API to enter Model-of-View instances and an unparse algorithm responsible for generating the target views.

The Template-of-Design (ToD) is the component of our generation approach that actual contains the hardware generator's logic. The ToD can be seen as a blueprint of all possible MoD instances. It is the part of the approach that takes the information provided by one or several Model-of-Things instances, processes it and generates one Model-of-Design according to the model contents.

The overall concept of our MDA implementation is covered to a higher level of detail in [7]. Moreover, [8] provides details on the design centric approach we take with our Model-of-Design. This paper focuses on the *view generation* aspect of our generation system. Although embedded into a three-layer MDA approach, our view generators can also be utilized independent of the rest of the MDA stack in existing environments.

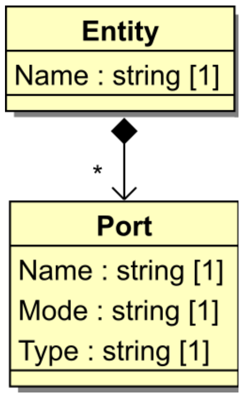


Figure 3. Simplified example of a Metamodel for a Model-of-View

IV. Our Approach to View Generation

This section introduces our novel approach to view generation. In the following, we first details how we our view generator looks and is utilized and try to convey how it facilitates the view generation task. We then show how we automatically generate the view generator and how it can be configured and extended using the so-called the *View Language Description (VLD)* which is used to generate the view generator.

A. Metamodel-based AST assembly

Our approach raises the level of abstraction of view generation. We do not directly generate target views through print-like statements or a template engine. Instead, we assemble a model of the target view. For this purpose, a Metamodel that captures the structure of language underlying target view is generated. In the MDA flow we describe in section II, this Metamodel acts as Metamodel of the Model-of-View, the model assembled is therefore the Model-of-View.

```
1  def buildAdderEntity(inPortWidth, Addends):
2      e = Entity(Name='ADDER')
3      for i in range(Addends):
4          e.addPort('PORT%d' % i, 'in',
5                    'std_logic_vector<%d downto 0>' % inPortWidth)
6
7      sumWidth = inPortWidth + int(ceil(log(Addends, 2)))
8      e.addPort('SUM', 'out', 'std_logic_vector<%d downto 0>' % sumWidth)
```
Listing 2. Snippet of Code generating a VHDL entity using our auto-generated API

Based on the Metamodel of the Model-of-View, a metamodeling environment provides an API that allows filling and accessing the API from a programming language environment. The process of view generation is thus reduced to a model-to-model transformation from a more abstract model to a more precise model (in our MDA stack, this is the transformation from Model-of-Design to Model-of-View).

Instead of using print-like statements or template languages, we now use more abstract, more readable Python code to generate the view. Figure 3 shows a simplified Metamodel for a VHDL Model-of-View. The API a Metamodeling environment generates is fairly straightforward and can be reproduced looking at Figures 1b) and c). Listing 2 sketches how view generation looks like using our approach. The snippet contains an example of a code generator which utilizes this API instead of a template engine or print-like statements. The snippet contains one Python function `buildAdderEntity`. This function is called with different integer arguments `inPortWidth` and different integer arguments `Addends`, both of which can be extracted from other, more abstract models. The code first instantiates an `Entity` in the Model-of-View and provides its `Name` attribute. Depending on the function's arguments, it then creates Model-of-View instances containing different port types and a different port count. For this purpose, a for-loop is inserted in lines 3 to 5. The number of iterations is controlled by the input parameter `Addends`. Every loop iteration adds a new `Port` to the Model-of-View. Lines 7 and 8 eventually compute a meaningful size for an additional port and add it to the Model-of-View. Much of this could of course also be solved without using a generation based approach. The purpose of this example is only to illustrate the generic process; it does not cover the large amount of possibilities generation-based approaches offer.

The main contribution of our view generator is the automated yet configurable path from the Model-of-View to the generated target code. The necessary input to provide this automation and to automatically generate the Metamodel for the Model-of-View is the *View Language Description (VLD)*. In the following, this description is detailed.

B. Background: Extended Backus-Naur Form and Abstract Syntax Trees

We exploit that all views we target with our code generation adhere to formal grammars. Using descriptions of these grammars, it is possible to describe the set of all syntactically correct codes of a certain target language. The most commonly used syntax to describe the grammar of formal languages is the Extended Backus-Naur Format (EBNF). EBNF descriptions are usually available for most common languages, often even incorporated in the standards defining the languages. Listing 1 shows an example of the EBNF of Verilog HDL. An EBNF description basically consists of a set of production rules (`conditional_statement` and `statement_or_null` in the example provided). These rules are assigned to a series of symbols, either production rules of their own or terminal symbols. EBNF further several special symbols such as `[]` brackets for optional sequences and `{}` brackets for sequences that may appear zero to many times. The vertical bar character `|` can be used to express alternation. A valid Verilog expression can therefore either consist of a primary or of a `unary_operator` followed by a `primary`, or of a `string`, etc.

```
conditional_statement ::= 'if' '(' expression ')' statement_or_null
                          [ 'else' statement_or_null ]
statement_or_null     ::= statement ';'
expression            ::= primary | unary_operator primary | string |
                          expression binary_operator expression |
                          expression '?' expression ':' expression
```
Listing 1. Snippet from Verilog 2001 EBNF Syntax

```
Entity ::= 'ENTITY ' <Name> ' IS\n' [Ports] 'END ' <Name> ';\n';
Ports  ::= $indent('\t')$('PORT(\n' $indent('\t')$(+Port%[0:-2]: ';\n';
                                                          [-1]  : '\n'  %+) ');\n');
Port   ::= <Name> ' : ' <Mode> <Type> ;
```
Listing 3. Simplified Snippet from View Language Description of the VHDL language

Aside from defining what a valid view looks like, the grammar also constrains the hierarchical structure of possible, syntactically correct views. Parsers for languages usually transform the view content into a so-called Abstract Syntax Tree (AST) representation. The AST is a data structure which can be best described as an object tree that describes the view, however strips redundancies and elements which are irrelevant for processing of the view (e.g. indentation). It captures the essence of what a view provides. This representation is subsequently used by the compiler for the set of optimizations and transformations it performs.

C. *View Language Description and Generation Process*

Our approach to view generation uses a syntax description similar to EBNF, the so-called *View Language Description (VLD)*. Using this description, we automatically generate a view generation framework consisting of

1) the Metamodel of the Model-of-View
2) the API derived thereof
3) the unparse mechanism used to turn the AST-like model into target views

Similar to EBNF, our View Language Description relies on the definition of a set of production rules. The production rules defined in the code snippet of Listing 3 are `Entity`, `Ports` and `Port`. These construction rules are also made up of terminal and non-terminals (production rules of their own). They may further utilize EBNF's set of special symbols as well as new symbols, attributes and conditional formatting directives we introduce to provide missing functionality.

To understand the motivation behind the artifacts we introduced into our View Language Description, it is important to note that the EBNF description in Listing 1 is only a valid description of the Verilog language under the agreement that any whitespace characters can be inserted between terminal or rule names, providing some freedom on how source code is formatted. While this is useful from a language and parsing perspective, it introduces abstraction. One instance of an AST does therefore not describe exactly one view. Instead, it removes details such as formatting.

It is further important to emphasize the two main requirements when designing a View Language Description: the automated generation of intuitive APIs and the automated generation of configurable target code formatting. The following shows the additional artifacts we introduced into the View Language Description to meet this goal and details why they are helpful.

Automated generation of intuitive APIs: A Metamodel has to be automatically created from the View Language Description. It must have a shape that guarantees that the API built with the Metamodel description is intuitive to use when automatically populating it through view generator code. To meet this requirement, we define a mapping from a VLD to a Metamodel. For every production rule, we introduce a class into the Metamodel. For every non-terminal in that production rule, a composition onto the respective class is created. Our VLD has a lower level of granularity than EBNF: Instead of e.g. defining identifiers as a concatenation of legal alphanumerical characters, we introduce attributes into the VLD. These attributes can contain legal values of common Python types. For every attribute in the production rule, an attribute is added to the Metamodel class. An attribute `<Name>` in the production rule `<Entity>` for example introduces a string attribute in the Metamodel class Entity, a specification of types other than string is possible using `<Count:int>` syntax. This allows us to directly introduce variables such as names or integer values from the host language into the Model-of-View from the view generator's API.

We further added a +…+ construct into the VLD. This symbol describes one or more occurrences of the inboard

```
name_list ::= +<name:string> ', '+
name_list ::= <firstName:string> ', ' {<otherNames:string> ', '}
```
Listing 4. Illustration of the use of the +…+ symbol in contrast to {…}

```
name_list ::= +<name:string> %[0:-2]: ', ';
                                [-1]  : '\n' %+
```
Listing 5. Illustration of the use of the %…% formatting construct

attribute or production rule. EBNF usually resolves this by the concatenation of one mandatory occurrence and one occurrence of zero or more times. Listing 4 shows how a rule `name_list` can be defined in two different ways, without changing the grammar of the view's language. The first approach simply introduces one attribute `name` of multiplicity one or more into the `name_list` class of the Metamodel, which provides the desired API for the view generator. The second approach introduces two attributes into the class, one with multiplicity one, named `firstValue`, and the other with the multiplicity zero or more, named `otherValues`. This means the syntactically different values in the list are treated differently although they are semantically identical. This is undesirable for two reasons. First, the Metamodel should reflect the semantics of the view and automatically provide the correct syntax. Second, introducing two attributes makes writing view generators more difficult. In our example, a view generator implemented on top of our framework would have to make sure that the first value is stored in `firstValue`, all others are stored in `otherValues`. In the example provided by Listing 2, this would for example require checking for `i == 0` inside the loop and then to either use the `addFirstPort` method or the `addOtherPorts` methods. The `+…+` construct eliminates the need for the developer to do this check manually.

Automated generation of configurable target code formatting: The VLD has to be sufficiently precise to create source code which follows any formatting and indentation convention. It is thus necessary to specify whitespaces and indentation characters. This is addressed through the introduction of special formatting directives.

The first of these directives addresses a problem introduced through our `+…+` construct. The two production rules we suggested for `name_list` would both generate a target view where each name – including the last one – is followed by a comma character. This illustrates a frequent problem in common view languages: the terminal strings surrounding the first or the last instance of a repetitive sequence often differ from the rest of the sequence. Listing 5 shows how the special formatting construct (inside of `%…%`) can be used to omit the comma following the last name and to introduce a line break instead. Using the array index notation in the listing, any terminals can be modified for arbitrary positions. Listing 2 also illustrates this in the VHDL entity example, where all ports but the last one are followed by semicolons.

The remaining issues with formatting include correct and configurable indentation, alignment of certain positions in subsequent lines and line breaking at a certain line width. We solve all these issues using a modular approach. For each issues, we defines special functions, which we call postprocessing routines. The routines can be applied to code generated by certain production rules using by wrapping them with `$indent('\t')$`, where indent is the name of a certain postprocessing function and `'\t'` would be an argument passed to it.

The postprocessing routines are implemented in Python and can be utilized across different view languages. It is further possible to build custom postprocessing routines for additional formatting needs. The indent postprocessor inserts the character that is passed as argument in front of every line of the output generated by the symbols it encloses.

Our format postprocessor uses a set of multiple tab-key symbols, each provided as argument. Each of these symbols may only occur once per line in the view code it processes. It treats each of those tab characters individually and modifies the lines that contain the character. For each of those lines, whitespaces are inserted so that the characters following the tab-key symbol are all aligned one below the other after the postprocessing step.

Instead of simply describing those rules as a composition of terminal and non-terminal symbols, we introduce additional formatting commands, for example to control indentation (e.g. `$indent()$`) and conditional terminal symbols. To sum up, we introduce the (`+…+`) notation in addition to optional components in square brackets (`[…]`) and optional repetitions in curly brackets (`{…}`), allowing a VLD to require one or more occurrences of a set of inboard symbols. Special formatting constructs (inside `%…%`) allow us to create different output for e.g. the first or the last occurrence of a repetitive item in the view.

```
ClassDecl        ::= 'class ' <Name> [ ': ' +<BaseClasses>%[0:-2]: ', ';
                                                          [-1]  : ' '  %+]
                     '\n{'
                     $indent('\t')$(  +PublicMember:ClassMemberDecl+
                                    ['private: ' +PrivateMember:ClassMemberDecl+] )
                     '\n}';
```
Listing 6. Simplified Snippet from View Language Description of the C++ language

```
ModuleDecl ::= 'SC_MODULE(' <Name> ') \n '
               '\n{'
               $indent('\t')$( +Port:PortDecl+
                               Processes
                               +PublicMember:ClassMemberDecl+
                               ['private: ' +PrivateMember:ClassMemberDecl+] )
               '\n}';
Processes ::=  + 'void ' <Name> '(); \n' +;
```
Listing 7. C++ View Language Description extended with SystemC artifacts

V. Advanced Considerations

Our experience shows that the quality of the generation framework's API is the key to the applicability of our generation approach. To optimize this API, it was useful to make different derivations from the EBNF descriptions available for most languages.

Some languages provide a multitude of ways to do one particular thing. Quite often, it is considered good style to always do a particular thing in a certain way. Coding guidelines often require one of these options. For example, when providing a range for an unconstrained array VHDL, declarations like `std_logic_vector(31 downto 0)` and `std_logic_vector(0 to 31)` could be used for similar tasks. Instead of permitting both these options for port declarations, a View Language Description can enforce a certain encoding.
In addition to deliberately limiting the possible views to a subset of the syntactically correct target views, it is also possible and reasonable to introduce redundancy into the VLD. A VLD for SystemC provides a good example for this. While SystemC code could be generated with a view generation framework generated from a View Language Description of C++, it is possible to increase usability of the view generation framework through extending the C++ VLD.

Listing 6 shows a simplified example of a C++ VLD, describing class declarations. Listing 7 shows a snippet that can be added to the C++ VLD which adds support for direct instantiation of SystemC modules. When it is included, a developer can directly use something like `headerFile.addModule(…)` instead of calling `headerFile.addClass(…)` and setting the necessary base class for SystemC. This reduces the amount of boilerplate code the developer has to manually write when using the view generator. Moreover, it can be used to enforce constraints. In the given example, using the `Processes` rule, the VLD asserts that process methods may not have any arguments.

VI. Results

We presented in this paper a novel approach to generating code by generating a generation framework from an EBNF-like notation. Our approach can be applied to a wide variety of computer languages – such as C code for firmware development, RTL languages and other hardware related formats (e.g. VHDL, (System)Verilog or UPF). It can further be used to generate documentation in formats such as TeX or DITA.

Our approach to view generation has several advantages over the traditional approaches, addressing all the issues we identified in Section II of this paper:

- Improved readability compared to traditional view generators. Using our view generator, it is no longer necessary to mix target code snippets and generator code. This allows for the use of syntax highlighting, automated formatting and linting tools providing for the code generator language. This simplifies reuse and makes it easier to write correct code.
- Constraints on possible Models-of-View inherently guarantee that generated views are syntactically correct. While it is difficult to generate correct code using template engines or generation approaches using print-like statements; our approach can produce correct and well-formatted code at the first attempt.
- Components of the Model-of-View can be instantiated at any time during the execution of the code generator and before the first characters of the target view are unparsed. A developer can thus focus on the creation of his target view and does not have to take care of buffering and rearranging his generated results to insert them in the output code at the correct position.

While precise measurement of the productivity increase is difficult to perform, we observe a reduction in generator size by at least 55% for typical generators. While this is a strong indicator for productivity increase, it falls short to account for significant advantages of our approach and their impact on developer productivity and quality of the produced generators and views.

We experienced a strong increase in productivity, mainly supported by the information the generated Metamodels contain on types and multiplicities of individual attributes and compositions. Using the type hinting capabilities of Python, our metamodeling environment automatically embeds this information into the generated APIs. Modern IDEs can utilize this information to provide code completion and error detection, providing a level of assistance usually only available for statically typed environments while maintaining the flexibility of the dynamically typed Python environment.

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, Nov.-Dec. 2010. doi: 10.1109/MM.2010.81

[2] B. Nikolić, "Simpler, more efficient design," European Solid-State Circuits Conference (ESSCIRC), ESSCIRC 2015 - 41st, Graz, 2015, pp. 20-25. doi: 10.1109/ESSCIRC.2015.7313819

[3] Y. Lee et al., "An Agile Approach to Building RISC-V Microprocessors," in IEEE Micro, vol. 36, no. 2, pp. 8-20, Mar.-Apr. 2016. doi: 10.1109/MM.2016.11

[4] W. Ecker, M. Velten, L. Zafari and A. Goyal, "The metamodeling approach to system level synthesis," 2014 Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, 2014, pp. 1-2., doi: 10.7873/DATE.2014.324

[5] M. Bayer. The Mako templating system. http://www.makotemplates.org

[6] IEEE, "IEEE 1685: IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows"

[7] W. Ecker and J. Schreiner, "Introducing Model-of-Things (MoT) and Model-of-Design (MoD) for simpler and more efficient Hardware Generators" 2016 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC), Tallinn, Estonia, 2016, pp. 1-6., doi: 10.1109/VLSI-SoC.2016.7753576

[8] J. Schreiner, R. Findenig, and W. Ecker, "Design Centric Modeling of Digital Hardware," 2016 IEEE International High Level Design Validation and Test Workshop (HLDVT), Santa Cruz, CA, USA, 2016, pp. 46-52., doi: 10.1109/HLDVT.2016.7748254.

[9] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, “EMF: Eclipse Modeling Framework” Pearson Education, 2008.