A New Approach for Generating View Generators

Johannes Schreiner, Felix Willgerodt, Wolfgang Ecker
Infineon Technologies AG, Munich, Germany / Technische Universität München, Munich, Germany

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. In this area, 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: instead of directly generating target code with print-like statements, we suggest assembling an abstract model of the target view.

Our framework is built around this intermediate model which his similar to Abstract-Syntax-Trees. To assembly our defined intermediate format, **we automatically generate an API** with the help of metamodeling techniques. We also provide an **automatism to generate the final target view from the intermediate format**.

This reduces the view generation task to population of an intermediate using an API. 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.

View Generation State of the Art

How are view generators built today?
Software that traverses some data structures, computes information and prints to files (the generated views).

Problems:

- **Convoluted, hard to maintain code:** with increasing level of configurability, the readability of view generators suffers.
- **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.
- **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.

Metamodeling and Metamodeling Frameworks

In Metamodeling, every model is formalized by a so called Metamodel. These Metamodels define structure, constraints and other properties of models. Figures 1 and 2 illustrate this using a sample Metamodel and a model that adheres to this Metamodel.

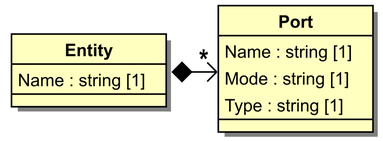

Figure 1 Simple Metamodel

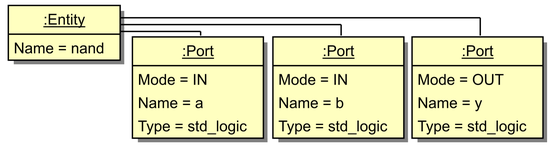

Figure 2 Model adhering the Metamodel of Figure 1

Metamodeling Frameworks utilize a formalized description of a Metamodel (e.g. a UML diagram) to provide a large set of features for working with models:

- **APIs to automatically populate and access models**
- Consistency checking of model data (i.e. constraint checking)
- Utilities for reading and writing to XML formats

Listing 1 shows a simplified **sample Python API for the Metamodel** from Figure 1. Using this API and other code generated by the Metamodeling environment, it is easy to work with models, read, write and modify them and to store them in a persistent way.

```
1 class Entity:
2   def getName()
3   def setName()
4   def addPort(Mode, Name, Type)
5   def delPort(pos)
6   def getPorts()
7
8 class Port:
9   def getMode()
10  def setMode()
11  def getName()
12  def setName()
13  def getType()
14  def setType()
```

Listing 1 Simplified API for Metamodel from Figure 1

Metamodeling is heavily utilized for all sorts of automation at Infineon. Commonly used metadata exchange formats such as IP-XACT area also based on the Metamodeling idea. Their success is a good example for the necessity of such well-defined, common sources of metadata.

Our Approach to Generating View Generators

Our approach to generating view generators relies on a Metamodeling framework and the API that is automatically generated by it.

We further utilize a description of the grammar, formatting and coding style of the target view. We refer to this as **"View Language Description (VLD)"**. This description is compact, independent of the generated view (only dependent on its language) and can thus be easily provided for all relevant view languages **in a one-time effort**.

The View Language Description provides:

- **Metamodel & View Generator API:** Using the View Language Description, a Metamodel for the so-called Model-of-View is derived. This description is used to derive a Metamodel and an API. This API (Front-end using Metamodeling Environment's API in Figure 3) is utilized by developers instead of manually generating target views.
- **Target code generator:** The entire fully automated process is pictured in Figure 3: Provided a view language description, our generation framework can automatically generate consistent, formatted and syntactically correct target, **for every model that is populated through the API**.

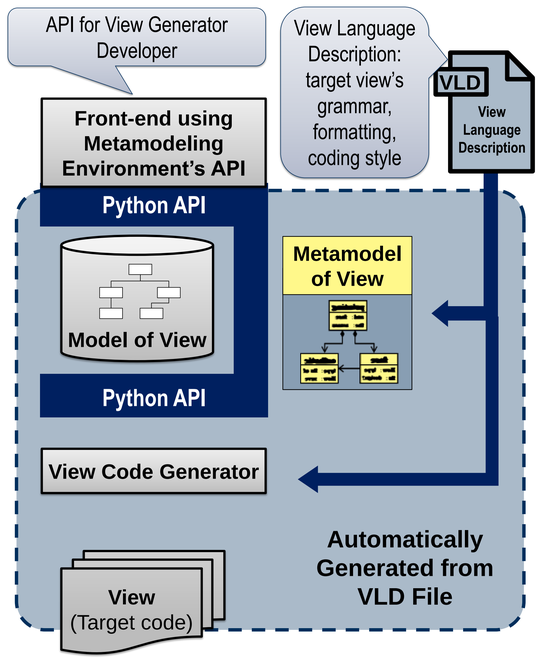

Figure 3 Architecture of View Generation Framework

The View Language Description

Listing 2 provides a simplified sample View Language Description describing the grammar of a **simplified VHDL Entity** and the **formatting of generated VHDL views**. When used to generate our view generator, the snippet from Listing 2 would result in the Metamodel in Figure 1 and an API similar to what is sketched in Listing 1.

```
Entity ::= 'ENTITY ' <Name> ' IS\n'
                [Ports]
           'END ' <Name> ';\n';
Ports  ::= $indent('\t')$(
'PORT(\n'
 $indent('\t')$(+Port%[0:-2]: ';\n';
                    [-1]  : '\n'  %+)
                  ');\n');
Port   ::= $align('!')$(<Name> ': '
                  '!a'  <Mode> '!b'
                        <Type>;
```

Listing 2 Snippet from View Language Description

Our VLD format contains a set of predefined routines for:

- Indentation `$indent('\t')$`
- Alignment `$align('!')$`
- Line Length Limitation

Additional routines for postprocessing/transformations of output can be easily included. The Implementation of these routines is straightforward in the Metamodeling Environments' host language.

Two different VLDs can describe:

- Different languages, resulting in a different Metamodel and different front-end API
- The same language, the same Metamodel and front-end API

→ Adoption and re-use for different coding styles without changes to the generator code

Results

- **Easier to write view generators:** We achieve a code size reduction by more then 50%, improved readability and faster to write templates
- **Syntax and formatting** of the generated code is automatically correct for every view generator as soon as the View Language Description is correct.
- **Models can be built in the intuitive order from generation perspective:** Developers can ignore the way view artifacts have to be printed and focus solely on the task of building a model that contains all necessary artifacts.