Abstract

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.

Metamodeling Frameworks

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.

The View Language Description

Listing 2 provides a simplified sample View Language Description describing the grammar of a simplified VHDL language, 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.

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-useable code for tasks such as indention 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.

Examples:

1. class Entity:
2. def getName():
3. def setName():
4. def addPort():
5. def delPort():
6. def getPorts():

Listing 1 Simplified API for Metamodel from Figure 1

Metamodeling is heavily utilized for all sorts of automation at Infineon. Commonly used metadada 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.

Results

- Easier to write view generators: We achieve a code size reduction by more than 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 view artifacts have to be printed and focus solely on the task of building a model that contains all necessary artifacts.