A Programmable Multi-Language Generator for CoDesign

J.P. Calvez, D. Heller, F. Muller, O. Pasquier

IRESTE, University of NANTES, FRANCE

Abstract

This paper presents an innovative technique to efficiently develop hardware and software code generators. The specification model is first converted into its equivalent data structure. Target programs result from a set of transformation rules applied to the data structure. These rules are written in a textual form named Script. Moreover, transformations for a specific code generator are easier to describe because our solution uses a template of the required output as another input. The result is a meta-generator entirely written in Java. The concept and its implementation have been demonstrated by developing a C/WxWorks code generator, a behavioral VHDL generator, a synthesizable VHDL generator.

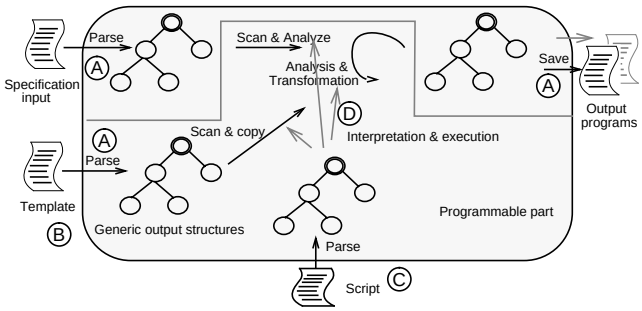
1: Goal

The CoDesign process includes the transformation of a high-level specification model into standard programming languages for microprocessors, DSPs and standard HDL for ASIC synthesis. One or several code generators are needed for that. Each generator must implement the know-how of professional Hw/Sw designers. A generator is strongly dependent on several aspects: the syntax and semantics of the specification input, the syntax and semantics of the code or program output, the transcription rules between the input and the output. Transcription rules are mostly versatile as they are related to the quality - simplicity, code size, speed, gate number, etc. - of solutions produced.

The question raised in this paper is: how to efficiently develop such generators? Most tools are based on a specific internal data structure representation of the input specification model from which target programs are produced. Script languages such as PERL and TCL were not found convincing for data structure manipulations. We first experimented a meta-tool [4], particularly GraphTalk from Xerox and its related meta-generator LEdit which is based on LEX and YACC. Because of several limitations we decided to conceive and implement an innovative global approach. This paper summarizes our solution and the concepts on which it is based. As a matter of fact, we have considered the MCSE functional model in its textual form as the input, but the solution is not limited to that model [1], [2].

2: Our solution: the concept of meta-generator

The principle of our meta-generator is depicted in Figure 1. We briefly explain the four aspects on which it is based.



-Figure 1 - Principle of our generation technique.

-A- Text <-> Data Structure conversion

The first aspect of our solution considers that an internal Data Structure (DS) organization can be automatically created from the grammar rules of the input text during its parsing. This means that each text must correspond to a grammar. This is naturally the case for standard languages such as C, C++, VHDL, etc. For specific texts such as specification models, their grammars must be defined - which is not a hard task and has to be done only once.

To convert any text into its equivalent DS and the reverse, we have developed the concept of meta-structure and have implemented two operations: **Load** to convert any text complying to a grammar into a data-structure; **Save** to generate and format a text from a data structure. Our solution is based on a meta-parser named JavaCup [3].

-B- Template as a model for the output

The second aspect of our solution considers a template model expressed in the syntax of the output language as another input to each generator. The template model (textual form) contains one instance of each program structure or construction needed in the text output. For example, for a synthesizable VHDL program, component entities and architectures, processes, blocks, configurations.... are needed. Each instance is declared in its most complex form. The powerfulness of this solution is that the output results from data structure operations such as copy or duplication

of parts of the internal data structure of the template, delete or update of the resulting data structure.

-C- Script concept for DS transcriptions

As the output is the result of data structure operations, the third aspect of our solution is an answer to describe all transformations needed. Rules on DS includes: scans and searches on the specification DS, searches of the template DS and copies of parts of it, transformations on the output DS. To describe any DS transformations we have specified a textual language named Script and its grammar.

The script language defines the required elementary operations on data structures and also defines how to build more complex transformation rules; the most complex one being the whole generator behavior.

The following list indicates the most significant elementary operations:

- Load a data structure from a text file according to a specific grammar,
- Save a data structure into a text file,
- Copy or clone a data structure from its designation,
- Copy a single node,
- Delete a whole data structure,
- Delete a single node,
- Update a field in a node,
- Add a node or a data structure (its reference) to a set of nodes.

More complex rules are based on the three usual rule compositions: sequential, iterative, conditional. An appropriate iterative instruction has been selected to efficiently operate on sets: *ForEach(Set : Rule)*;. Rules can also be imported from other scripts so enabling reusability. Rules can be declared in any order; therefore it is more a declarative language than a procedural one.

Constants and variables can also be declared to store and manipulate values, strings or grammar names. For an easy implementation of recursive rules, variables are managed as stacks and a set of operations on them are available, for example Push(Var), Pop(Var), LocalVisibility, IsIn, etc.

Thus, developing a generator consists in writing the appropriate set of transformations. A target program generation is obtained by the execution of the resulting script. In fact, our tool is really a meta-generator as the input and output texts are specified by grammars and its complete behavior is only resulting from the interpretation of a script.

-D- Interpreted and native execution of a script

The fourth aspect of the solution concerns the execution of any script. As a script is a text, it is first converted into a data structure with the Load operation. Then, its execution consists in the execution of an automaton able to scan the script DS and execute the elementary operations as procedures or methods. More powerful is the fact that a script DS can be internally and/or interactively modified to control the output produced. The script DS can then be converted into a textual form and re-executed. This can be an interesting means to create an interactive generation or synthesis tool. Moreover any script can be converted into a Java program to deliver a native generator. The result is a specific efficient generator.

3: Result and experimentation

We have prototyped 3 specific generators using the MCSE model as the specification input to demonstrate and validate the concept. Our tool currently runs on any platform and may also be used through the Web.

The three generators are not complete today and not fully debugged and tested. But we can give significant results which demonstrate the appropriateness of the concept and its implementability. The following table shows quantitative results for the 3 generators. File sizes are given in number of characters or bytes.

Generator	Script size	MCSE size	Template size	Output size	Interpreted Execution time	Java Code size	JAVA exec. time
CVxWorks	150 KB	6.33 KB	2.23 KB	3.2 KB	63 s	1 MB	34 s
VHDLSyn	194 KB	6.93 KB	18.6 KB	21.7 KB	111 s	1.2 MB	82 s
VhdlPerf	120 KB	5.41 KB	18.1 KB	33 KB	363 s	1.1 MB	138 s

The ratio between the Java code size and the Script size partly represents the speedup in development time for each generator.

4: Conclusion

Our meta-generator is fully portable as it is written in JAVA. Experiments of the technique by developing three generators demonstrate and validate our approach and its implementation. The scope of our tool is not limited to the goal considered in this paper. On one hand, the tool can consider any text as input and output if they comply with a grammar. It can also concurrently exploit several input texts (e.g. C et VHDL templates) and produce several code programs or models. On the other hand, the script can be easily modified or produced by another tool; it seems to us that it can be a promising solution to implement synthesis techniques for CoDesign. We are considering this aspect for Hw/Sw communication synthesis.

5: References

[1] J.P. Calvez, Embedded Real-time Systems. A specification and Design Methodology, John Wiley, 1993

[2] J.P. Calvez, A System-level performance model and method, CIEM, Issue #6: Meta-modeling: Performance, Software and Information Modeling, KAP Publisher, 1996, pp 57-102

[3] S.E. Hudson, JAVACUP: LALR parser generator for Java, User's manual, GVU Center, Georgia Institute of Technology, March 1996

[4] B. Nuseibeh, Meta-CASE support for method-based software development, Proceedings of Meta-CASE'95 Conference, Sunderland, UK, Jan 5-6 1995