Automatic Transformation of System Models in Automotive Electronics

Ralph Görgen[1], Jan-Hendrik Oetjens[2], Jan Freuer[2], and Wolfgang Nebel[3]

[1] OFFIS Institute for Information Technology Oldenburg, Germany
Ralph.Goergen@offis.de
[2] Robert Bosch GmbH Reutlingen, Germany
{Jan-Hendrik.Oetjens|Jan.Freuer}@de.bosch.com
[3] Carl von Ossietzky University Oldenburg, Germany
Wolfgang.Nebel@informatik.uni-oldenburg.de

Abstract. Evaluation and refinement of system models often require modifications in the model that follow concrete rules. In this work, a method for a flexible automation of such transformation steps will be presented. It allows savings in development time and reduces the error proneness. Therefore, a tool for rule based manipulation of VHDL design descriptions has been extended to enable its use with system models in C++ and SystemC. An automotive electronics application, the integration of SystemC modules into a MATLAB/Simulink simulation by automatic wrapper generation, will show its use in the design process.

Introduction

While electronic components in cars become more and more complex to optimize comfort, security, and environmental impact, the increasing pressure in the automobile market permanently involves cost reductions. A further increase of complexity comes with the special demands of automotive electronics, e. g. reliability and robustness over long periods of time in a harmful environment regarding to vibrations, temperature changes, and electro-magnetic interference. Design and verification of such systems are exceeding challenges that need support by appropriate methods and tools.

Often, the V-Model is the basis of hardware/software development processes. After the specification, an abstract model is created to perform first analyses. MATLAB/Simulink [15] and SystemC [14] are common in this context. Then, a stepwise refinement follows until a final implementation is found. The way from a system model to a final implementation requires lots of code transformations, either as a step to a lower level of abstraction, or to realize several design decisions. Some transformations need the creativity of a designer. Others follow concrete rules, they rather laborious work and a possible source of needless errors.

A cooperation of Robert Bosch GmbH and OFFIS has developed a framework for rule based design transformations[1]. It includes an easy way to define

[1] This work has been partially founded by the German Bundesministerium für Bildung und Forschung (BMBF) in the VISION project (01M3078D).

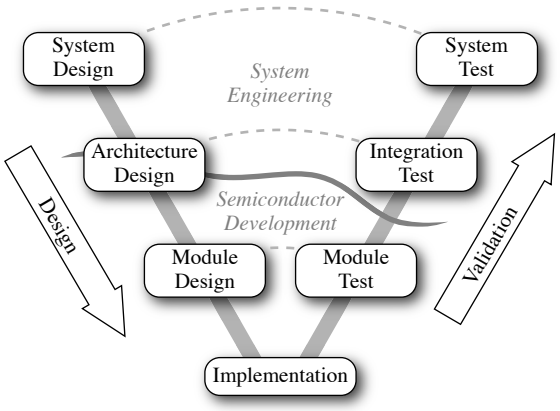

Fig. 1. V-Model.

transformation rules and to apply them automatically to a design description in the hardware description languages VHDL and VHDL-AMS [13]. In this contribution, an extension of the framework for C++ and SystemC is presented. Hence, the designer is able to use the same tool at higher levels of abstraction too. Furthermore, inter-language transformations become possible.

Section 1 explains the underlying design flow. In Section 2 the transformation framework is presented in detail. Section 3 deals with related work and shows the problems in the present case. In Section 4, the C++/SystemC frontend is described, and in Section 5, an automotive electronics example shows how the new possibilities can be used in the design process. Finally, in Section 6, the results are summed up and a short outlook on future work is given.

1 Transformations in a System Design Flow

Regarding to the V-Model (Figure 1), a system development process contains a number of more or less complex changes in the system description. Step by step, the final implementation develops from an abstract system specification. Some refinement steps require creativity of a designer or specific knowledge about the application. Automation of these steps is often impossible. Others offer very little degrees of freedom or follow concrete rules because they are defined by requirements or process standards or because they follow common laws. In general, performing them automatically is less error-prone and saves time and costs. Lots of these design steps can be realized with common tool chains but there are others that are not covered by available tools. Each application domain, company, or even particular designer knows individual design steps that are either too specific to be provided by commercial tools or based on confidential knowledge.

The method presented here allows the automation of individual design steps by creating a particular transformation rule. Whenever a design step is not covered by the available tools, the designer can decide whether its automation

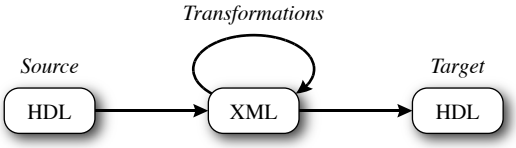

Fig. 2. Transformation Approach.

makes sense, depending on how often a transformation rule can be used and how complex its implementation is by contrast with the manual transformation. An important detail is that the design can be written out in a human readable and recognizable form. Hence, it is possible to perform pursuing steps by hand or by any other tool. The tool to create and apply the transformation rules is already in use and successful in the lower regions of the V-Model (VHDL). Currently existing applications are the insertion of clock gating cells to reduce power consumption, code obfuscation, insertion of hamming codes into busses, and many more. Now, it has been extended for its use at higher levels of abstraction.

2 Transformation Framework

The transformation framework allows the flexible transformation of design descriptions based on user-defined rules. Figure 2 shows the general transformation flow. At first, the design is read by a frontend and an XML based description is generated. Then, this XML tree can be transformed using XSLT [18]. When all transformations are finished, the result tree can be written out in the original description language. The tool is implemented in Java to allow its usage on every common platform. Furthermore, it is prepared for the integration into development environments, e. g. Eclipse [3].

Reading the Design Accordant with the flow mentioned above, firstly, it is necessary to create an XML representation of the design. This is done by a parser, which is generated with ANTLR [2], an open-source tool for parser and lexer generation. Its input is an EBNF like syntax definition extended by semantic attributes. The generated parser reads text in the particular language and builds an abstract syntax tree. Besides syntax elements, comments and formatting characters are included in that tree to allow its use in later processing steps. A tree generated like that, supplemented with some semantic information, and written out as XML is the basis of the actual transformations.

Transformation of the Design The transformation rules are implemented in XSLT. This is a language to create style sheets that describe how to generate an output document out of an XML input document. In substance, the transformation rules consist of two elements, code identification and code generation. The first determines which parts of the code should be transformed, the second which changes should be done with those parts.

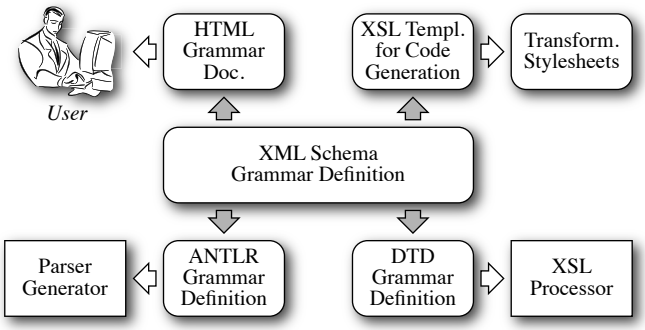

Fig. 3. Grammar definition as central document.

Output of the Transformed Source Text To write out the tree in its original description language after the transformation, XSLT is used again. A style sheet outputs the syntax tree elements as text. The original appearance of the code can be restored by considering formatting characters and comments as well. Hence, the designer is able to recognize the code and to continue to work with it.

Frontend Requirements

Some specific requirements result from the mentioned application scheme.

1. It must be able to create an XML tree.
2. In order to facilitate the implementation of transformation rules, the XML syntax tree has to meet the formal syntax definition as stated in the particular language standards as near as possible.
3. Besides the syntactical elements, the XML tree must contain comments and formatting characters. Their recovery has to be possible when transforming the design back to source code to preserve its readability.
4. The frontend should be maintainable and easy to extend; many description languages are permanently refined and the language standards are regularly adapted to new challenges too.

Extensions to Support Various Languages

The maintainability of the transformation tool's language support extensions is guaranteed by the use of a single central document, the grammar definition in form of an XML Schema (abr. XSD) [17]. As shown in Figure 3, any other components of the environment are created out of it automatically. An ANTLR grammar definition is built to generate the parser. The used XSLT processor does not support XSD. Therefore, a grammar definition as a DTD is necessary to validate the XML trees and therewith the syntactical correctness of the corresponding code. Furthermore, the XSD is translated to an HTML grammar documentation to relieve the implementation of transformation rules. Finally, an XSLT template is generated for each element of the XSD. These templates

are used by the transformation rules for code generation to create new XML elements and add them to the syntax tree. The use of XSLT templates generated out of the XSD ensures that the generated code is XSD conform and therewith syntactically correct. In conclusion, the main task for the integration of a new language is the creation of this central XSD grammar definition.

3 Related Work

This section presents other approaches to transform design descriptions and to parse C++ and SystemC and explains why they are inappropriate in our case.

Transforming Design Descriptions In industrial practice, it is common to use scripting languages like sed [4], AWK [1], and Perl [16] to modify design descriptions. They allow the definition of regular expressions to analyze the source code and perform changes to it. Languages like VHDL and SystemC are no regular languages and their analysis with such scripts is very limited. As a consequence, there are only local and not too complex changes possible.

In [10], another XML based representation of VHDL designs and the way to generate it is presented. Then, to the XML tree several transformations like the generation of HTML documentations can be applied. It is useful for VHDL but there is no support for any other language available.

Parsing C++ and SystemC One possible opportunity is the use of the frontend of a standard C++ compiler. Some of them offer a way to output their internal representation of the source code as XML trees [9]. That meets the first requirement but not the others. Conformance to the standard is not given. Only parts of the resulting trees correspond to the formal syntax definition on one hand, the here examined frontends add some compiler specific elements to the code on the other. Hence, requirement 2 is broken. But the crucial point is that requirement 3 cannot be fulfilled, the retrieval of the original source files is not possible. The problem arises from the two-stage strategy of common C++ frontends. At first, the source code is read by a preprocessor unit. Then, the preprocessed text is given to the actual C++ parser. The preprocessing contains the removal of comments as well as text replacements according to the preprocessor directives. The XML output of the compiler frontends solely contains the so prepared C++ code. Consequently, some information like the original file structure, comments and formatting characters are not included in the result tree and thus unrecoverable.

Furthermore, there are explicit SystemC parsers like KaSCPar [5] or Pinapa [11]. KaSCPar performs two phases. The first creates a syntax tree. Within the second, the generated tree is elaborated and supplemented with structural information. In contrast to C++ parsers, the KaSCPar frontend knows dedicated tokens for some of the SystemC specific Keywords. This facilitates recognizing SystemC constructs in the syntax tree but it complicates maintenance and adaption to

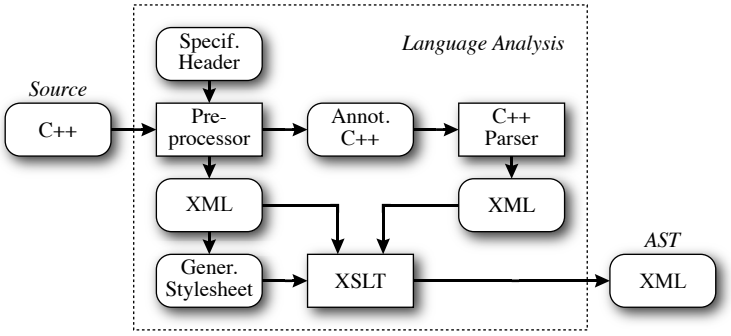

Fig. 4. Frontend work-flow.

changes in the language standard. That conflicts with requirement 4. The open-source solution Pinapa also provides a SystemC frontend that supplements the syntax tree with structural information in a second step. It uses a tree that is similar to the internal representation of the GNU GCC. The tool does not offer a possibility to output the tree in XML, and hence, requirement 1 is not met. Both of the two parsers break requirement 3. Since they both use the GNU GCC preprocessor, important information for the recovery of the source text is missing in the result trees.

Thus, to the best of our knowledge, none of the existing approaches is able to satisfy all of the key requirements identified in Section 1 for our specific application scenario.

4 C++/SystemC Extension

The C++/SystemC frontend developed by us works in three phases. In the first phase, a customized preprocessor is called, in the second the actual C++ parser. In the third phase, the generated C++ syntax tree is transformed to a modified syntax tree with XSLT. Figure 4 pictures this procedure that will be explained in detail in the following section. Due to the fact that the frontend is based on a regular C++ frontend, it is easy to extend it for other C++ based languages like SystemC-AMS [6] or OSSS [7]. One solely has to integrate the particular header files.

To reduce the effort for the development of the C++ parser and preprocessor, we modified existing implementations in ANTLR for our purpose. The preliminaries originate from Youngki Ku (preprocessor) and David Wigg (C++ parser). Their results are available for free at the ANTLR homepage [2].

Preprocessor

As described in Section 3, it is important in our application that code modifications done by the preprocessor are reconstructible to allow reassembling the code

in its original appearance. Therefore, comments and formatting characters are not removed by the preprocessor described here. But the preprocessor directives are processed as usual. That means in substance expanding macros, integrating include files and evaluation of conditional compilation directives. For this reason, a preprocessor syntax tree is created and then the prepared C++ code is generated out of it. In addition to these basic functionalities, every statement concerning the preprocessor is annotated with a unique ID. Meta-tokens that contain these IDs are inserted in the generated C++ code. They mark the parts of the code that arise from the corresponding preprocessor action. As a result, the output of the preprocessor is clearly associated with the original source code.

Finally, the preprocessor result and its syntax tree are passed to the downstream C++ parser.

C++ Parser

The C++ parser receives the preprocessor output and converts it to a C++ syntax tree. Therewith, comments and formatting characters are included in the tree. Since they do not have any syntactical meaning, no specific nodes are generated for them. They are added to the following terminal as attributes. The meta-tokens inserted by the preprocessor do not become nodes in the C++ syntax tree but attributes to terminals as well. Figure 5 shows schematically the connections between input text, preprocessor syntax tree, preprocessor output and C++ syntax tree. Macro definitions and macro calls can be seen in the preprocessor syntax tree. Each macro call is annotated with a unique ID and a reference to the corresponding macro definition. In addition, the ID of the macro call is inserted as a meta-token into the preprocessor output text. The C++ parser can use this ID as a reference to the macro call to annotate it to the terminals in the C++ syntax tree. Finally, the syntax trees of preprocessor and C++ parser are written together into a single file as XML trees. Now, this file contains all necessary information. The C++ syntax tree contains the actual relevant code as well as comments and formatting characters, the preprocessor tree contains the original source code. IDs and references describe the relations between the two trees.

Transformation to Modified Syntax Tree

In the third phase, the generated XML file is processed with XSLT. Here, a difference is made between the actual design and code parts that have been added by the integration of external header files. This differentiation is crucial because external code must not be modified on one hand and it does not need to be written out on the other. Usually, these headers are system files or in the case of SystemC part of the language definition. Those parts are allowed to be removed because they do not contain any relevant information. A further task in this phase is to undo the calls of those macros whose definitions are in one of the external files. This step facilitates the recognition of specific parts of the code because macros are often used as language elements, e. g. `SC_MODULE`

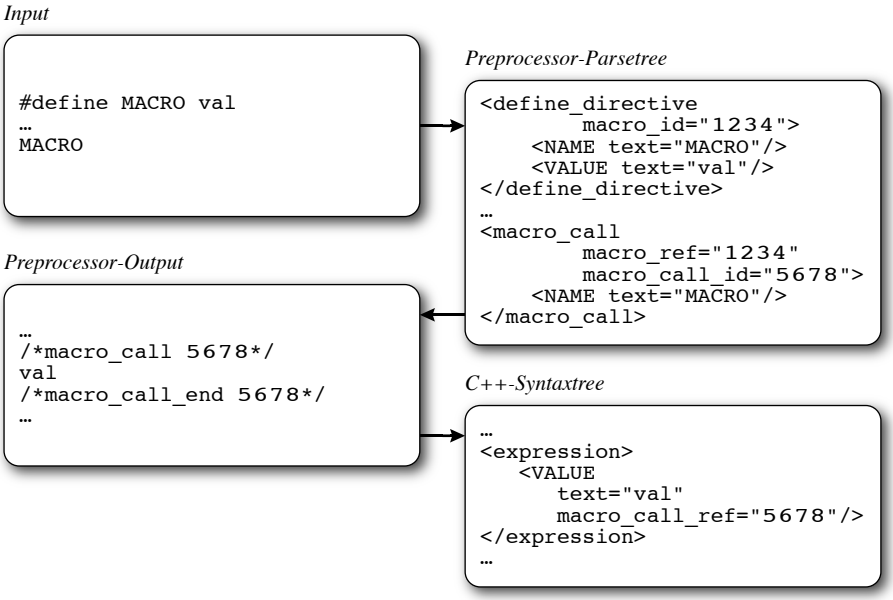

Fig. 5. IDs and references in the syntax tree.

or `SC_METHOD`. This procedure automatically works with any other library like OSSS or SystemC-AMS as well. The result is a modified syntax tree and any transformation can be applied to it.

Output of the Transformed Source Text

To get back to the original source text, XSLT is used again. The XML tree is written to files as C++ code. The original formatting and directory structure is reconstructed as far as possible. The preprocessor logs the beginnings and ends of each file as well as include directives and file paths. With this, the reconstruction of files and directory structure is unproblematic. The retransformation of macro expansions and conditional compilation directives is more complex. If the code that came out of such a directive has been changed by a transformation and new or transformed code has been generated, the original code must not be written out. As a consequence, the original formatting and the particular preprocessor directive get lost. All parts of the code that has not been transformed are typed out exactly like the preprocessor read it. For newly generated parts that do not contain formatting information a standard formatting is used.

A special case of conditional compilation directives is the surrounding `#ifndef` – `#define` – `#endif` which is used in header files to prevent multiple compilations. When this instruction sequence cannot be reconstructed, a new one that surrounds the content of the file is generated automatically.

Now, the transformation flow is complete. Design descriptions in C++, and as a consequence in SystemC as well, can be read, transformed and typed out again as source text, and the frontend meets all requirements stated in Section 1.

5 Example: Generation of SystemC Wrapper Modules

In the following section, we want to show how the tool can be used in the design flow. The automatic generation of wrapper modules to connect SystemC modules to a MATLAB/Simulink model is used as an example. Despite the simplicity of the example, it involves a significant gain in efficiency. The automation of this development step saves time and the error probability is reduced. The generated wrappers are part of an extended testbench concept that is used for the development of automotive electronic components. The following subsection illustrates it in short, before the wrapper generation itself is explained.

Extended Testbench Concept

In support of verification by simulation, an extended testbench concept has been developed at Robert Bosch GmbH [8]. It is shown in Figure 6. Its main purpose is to use the same testbench modules at multiple levels of abstraction and in several simulation environments. The design under verification may be present as MATLAB/Simulink, SystemC, or VHDL model. The same testbench modules can be used in all of the three cases. We want to have a closer look at the coupling of a MATLAB/Simulink simulation with SystemC testbench modules now. Using MATLAB/Simulink and SystemC at the same time causes problems because the two simulation environments are based on two different simulation concepts. MATLAB/Simulink uses continuous simulation, SystemC discrete event simulation. It is required to synchronize the two environments in an appropriate way. Furthermore they use different data types so that conversions are necessary for data exchange. The data type conversion as well as the synchronization is done by a SystemC wrapper that encloses the actual testbench module. This wrapper is composed of two parts. Firstly, the wrapper module must be derived from an abstract wrapper class. It must provide one port for each port of the testbench module. Secondly, a so called `createModule` method must be implemented to bind the ports. This method also registers the ports because the wrapper must know the number of ports and whether they are input or output ports.

Automatic Wrapper Generation

Now, the SystemC wrappers should be generated automatically by a transformation rule. At first, the frontend must read the testbench module and transfer it to an XML syntax tree. Then, the required parts are inserted. That is, the tool generates subtrees that correspond to the wrapper module and the `createModule` method. For each port of the testbench module the actual port declaration in the wrapper module and its binding and register statements in the `createModule`

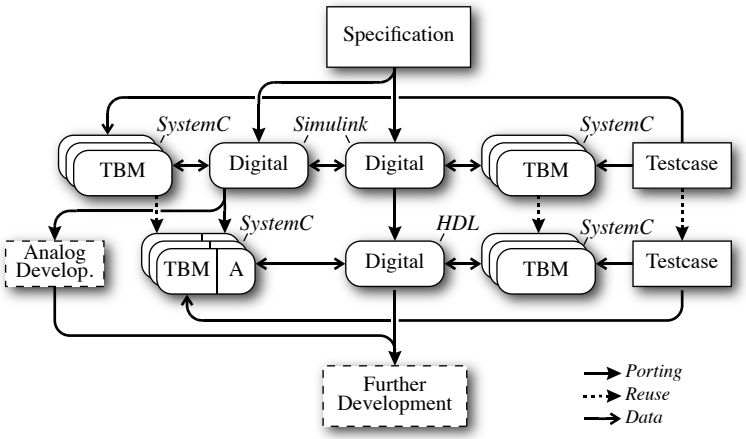

Fig. 6. Extended Testbench Concept.

method are generated. Then, these subtrees are inserted into the syntax tree at the appropriate positions. Finally, the extended syntax can be written out as C++ code and the testbench module is ready to use it in a MATLAB/Simulink simulation. Since an example implementation of a wrapper was available in our case, the first step was to set up a transformation rule that generates exactly this source code. That is very easy because the transformation tool offers a feature to read a piece of source code and automatically create a rule that generates the same code in form of a syntax tree, which can be inserted in a design then. In this rule, those parts where the port declaration, binding, and register statements are generated had to be surrounded by loop statements to generate the required elements for all ports of any module. Furthermore, the static port names had to be changed to dynamic ones picked out of the testbench module.

Evaluation of the Results

By means of the transformation tool, its C++/SystemC extension, and the transformation rule, we are able to surround any SystemC testbench module with a wrapper to integrate it in a MATLAB/Simulink simulation. This can be done fully automated and many times faster as well as less error-prone than its manual implementation. Table 1 compares the durations for the manual and automatic wrapper generation to test of a sensor evaluation circuit used in an automotive controller IC. When the manual wrapper generation is used, the implementation of the wrappers can start immediately. Otherwise, it is necessary to implement the code for one wrapper at first. Then, the creation of the transformation rule can start as depicted above. After that, the generation of further wrappers is possible in only a few seconds. The times used here had been worked out with a concrete example. Of course, the exact values heavily depend on the particular testbench module and designer. But, it is obvious that the automatic generation takes longer when only few wrappers are needed. However, the transformation rule must be implemented only once and can be used again and again. As shown

Table 1. Comparison of wrapper generation.

Task	manual	automatic
Implementation 1st wrapper	1 h	1 h
Transformation rule	–	2 h
Implementation 2nd wrapper	1 h	–
Generation 2nd wrapper	–	≈ 0 h
Total	2 h	3 h

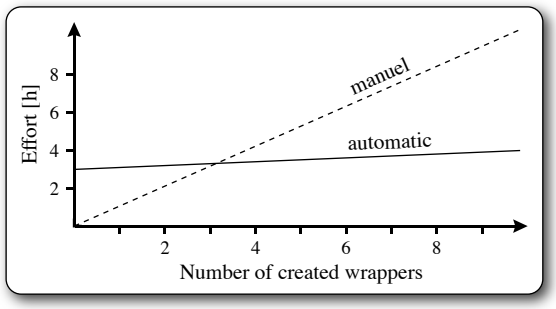

Fig. 7. Time needed for creation of the wrappers.

in Figure 7, the time needed for the generation of several wrappers rises very little whereas the effort for the manual method increases much more. Since generally several testbench modules are used in one project, and with it, several wrappers are needed, and additionally, the same transformation rule can be used in more than one project, the automation of this design step is a good opportunity to improve the development process in terms of design effort and quality.

Besides transformations of SystemC models, the newly added grammar definition allows inter language transformations. An application that uses this functionality is a VHDL-to-SystemC translation that already has been published [12]. Additionally, this example shows the potential of our approach.

6 Conclusion and Outlook

This contribution presented an extension for a tool for design transformations. Apart from its use with design descriptions in VHDL, it can be applied to C++ and SystemC. As a result, the developer can use a tool he already knows in the context of VHDL as well at higher levels of abstraction. He is able to read descriptions in C++ and SystemC, transform them, and output them again. In substance, the extension consists of a C++ preprocessor and parser. They are based on existing implementations available for free. As a result, the effort for the adaption to our needs was very small. An example has shown how the new possibilities can be used in a design process. In order to do so, an automatic generation of SystemC wrappers has been implemented. It allows the integration

of SystemC modules into a MATLAB/Simulink simulation. With the automation of that step, it is possible to achieve results faster and less error-prone.

In the future, we plan to use the tool for more complex transformations, e. g. optimization of design descriptions in SystemC. Additionally, we want to integrate Verilog as a further design language.

References

1. Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Addison Wesley, 1988.
2. ANTLR. ANother Tool for Language Recognition. http://www.antlr.org, 2009.
3. Eclipse Foundation. http://www.eclipse.org, 2009.
4. Free Software Foundation. ses, a stream editor. http://www.gnu.org/software/sed/manual/sed.html, 1999.
5. FZI Karlsruhe. KaSCPar - Karlsruher SystemC Parser Suite. http://www.fzi.de/sim/kascpar.html, 2006.
6. Christoph Grimm, Martin Barnasconi, Alain Vachoux, and Karsten Einwich. An Introduction to Modeling Embedded Analog/Mixed-Signal Systems using SystemC AMS Extensions. Whitepaper, Open SystemC Initiative, 2008.
7. Cornelia Grabbe, Kim Grüttner, Henning Kleen, and Thorsten Schubert. *OSSS - A Library for Synthesisable System Level Models in SystemC*, 2007. http://www.system-synthesis.org.
8. Kai Hylla, Jan-Hendrik Oetjens, and Wolfgang Nebel. Using SystemC for an extended MATLAB/Simulink verification flow. In *FDL '08: Proceedings of the Forum on Specification and Design Languages*, 2008.
9. Kitware. GCC-XML - XML output for GCC. http://www.gccxml.org, 2007.
10. T. Karayiannis, J. Mades, André Windisch, T. Schneider, and Wolfgang Ecker. Using XML in VHDL Analysis and Simulation. In *Proceedings of the Forum on Design Languages (FDL)*, September 2000.
11. Matthieu Moy. Pinapa: An open-source SystemC front-end. http://greensocs.sourceforge.net/pinapa/, 2005.
12. Jan-Hendrik Oetjens, Ralph Görgen, Joachim Gerlach, and Wolfgang Nebel. An Automated Flow for Integration Hardware IP into the Automotive Systemc Engeneering Process. In *DATE '09: Proceedings of the conference on Design, automation and test in Europe*, 2009.
13. Jan-Hendrik Oetjens, Joachim Gerlach, and Wolfgang Rosenstiel. Flexible specification and application of rule-based transformations in an automotive design flow. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, 2006.
14. OSCI. IEEE Std. 1666, SystemC Language Reference Manual. http://www.systemc.org, 2005.
15. The Mathworks Inc. http://www.mathworks.com, 2009.
16. Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly Media, Inc., 2000.
17. World Wide Web Consortium. XML Schema 1.0. http://www.w3.org/XML/Schema, 2004.
18. World Wide Web Consortium. XSL Transformtions (XSLT) Version 2.0. http://www.w3.org/TR/xslt20, 2007.