An Automated Flow for Integrating Hardware IP into the Automotive Systems Engineering Process

Jan-Hendrik Oetjens
Robert Bosch GmbH
AE/EIM3
Postfach 1342
72703 Reutlingen
Jan-Hendrik.Oetjens@de.bosch.com

Ralph Görgen
OFFIS Institute
R&D Division Transportation
Escherweg 2
26121 Oldenburg
Ralph.Goergen@offis.de

Joachim Gerlach
Robert Bosch GmbH
AE/EIM3
Postfach 1342
72703 Reutlingen
Joachim.Gerlach@de.bosch.com

Wolfgang Nebel
Carl v. Ossietzky University
Embedded Hardware-/Software-Systems
26111 Oldenburg
nebel@informatik.uni-oldenburg.de

Abstract

This contribution shows and discusses the requirements and constraints that an industrial engineering process defines for the integration of hardware IP into the system development flow. It describes the developed strategy for automating the step of making hardware descriptions available in a MATLAB/Simulink based system modeling and validation environment. It also explains the transformation technique on which that strategy is based. An application of the strategy is shown in terms of an industrial automotive electronic hardware IP block.

1. Introduction

The domain of automotive electronics system design is characterized by specific constraints and environmental conditions that significantly differ from those in other design domains like mobile communications, multimedia or high performance computing. This results from the fact that automotive electronics systems are often involved in safety critical car functions that have to be highly reliable and robust over a long lifetime, and are used in harsh environments in terms of vibrations, temperature changes, or electromagnetic interferences. These specific constraints have to be considered during all steps of the automotive electronics systems engineering process. This contribution describes a methodology developed at the semiconductor division of Robert Bosch GmbH in cooperation with OFFIS Institute for Information Technology, which provides a seamless and automated strategy for making intellectual property (IP) hardware models available on higher levels of model abstraction. This allows considering hardware IP accurately during the system modeling and exploration phase. Hence, the identification of incorrect system behavior or problems that may arise during system integration is possible in an earlier stage of the systems engineering process.

Most of the companies' system development processes follow – at least, in their basic principals – the well-known V model [1]. In our specific case, there is running an additional boundary through that V model. It partitions the process steps into tasks to be done within our semiconductor division and tasks to be done within the corresponding system divisions (see Figure 1). The prior mentioned acts as a supplier of the hardware parts of the overall system for the latter. Such boundaries also exist in many other companies whereas the exact boundary shape in the V model may differ. In our case, the system divisions are responsible for the engineering of the system application, the definition of the coarse grain system architecture and the assembling of the system by integrating all components (mechanical parts, sensors, hardware IP, software, etc.). An aspect that arises from that partitioning is that both sides, systems engineering and semiconductor development, have different views on the system and its constraints, different understandings and know how, and different design methods, tools, and environments. This makes a seamless interaction of both sides an ambitious and highly important challenge.

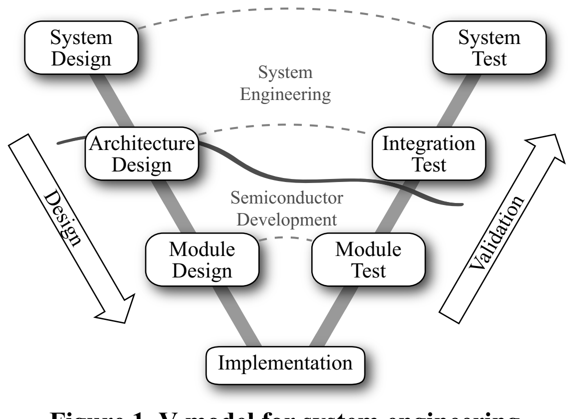

Figure 1. V model for system engineering

In automotive electronics system design, the modeling and validation of heterogeneous system behavior on a high level of abstraction (at the systems engineering level) is often done using MATLAB/Simulink. For the design of the integrated circuits (at the semiconductor development level), traditional register-transfer level hardware description languages like VHDL and Verilog play a major role. Regarding the "design" arc of the V model a typical strategy for crossing the boundary between systems engineering and semiconductor development can be divided in two steps. Firstly, a natural language specification document is derived for the system parts to be implemented in hardware from the Simulink model (to be done at the systems engineering level or in tight cooperation of both levels). Secondly, engineering of the hardware component is started (at the semiconductor development level) based on this document. Vice versa – regarding the "validation" arc of the V model – the hardware component can be considered at systems engineering level at the earliest after the whole hardware development task has finished and all system components are available for their integration in a physical prototype. This means that a more accurate Simulink model of the developed hardware IP, which would allow starting the system validation task already at model level, does not exist. This lack becomes more dramatically in the context of current system-level design methodologies like platform-based design or component-based design. Therefore, an efficient reuse of existing hardware IP is highly crucial.

The paper is organized as follows: In section 2, existing strategies for implementing cycle-accurate Simulink models are discussed. Section 3 outlines related work in the automatic generation of executable models from HDL code. Section 4 presents our approach using a conversion into a SystemC model and its integration into Simulink. The described approach was used to integrate an automotive hardware IP into Simulink, which is presented in section 5. Finally, section 6 concludes with an outlook on future work.

2. Existing strategies

Existing strategies to overcome this lack include the co-simulation of hardware descriptions and Simulink models at systems engineering level. This is expensive in terms of effort and costs because the expertise and environment for simulating hardware does usually not exist on that level. In case of external customers, another aspect is that co-simulation usually requires the source code of the hardware description, which might be critical due to non-disclosure reasons. Other strategies include the manual generation of C code descriptions for the hardware functionality to be embedded into Simulink via S-functions. In practical use, this is often not feasible due since the manual coding step is time-consuming and error-prone. Additionally, the effort to keep the hardware description and the Simulink model compliant (e.g., after several adaptation or configuration steps on hardware side) is quite high. Therefore, this step is often skipped in practical use due to the tape-out schedule, which is a handicap for an efficient reuse.

3. Related work

Because co-simulation and manual generation of C code are often not applicable in practical use, an approach for the automatic generation of C/C++ code from hardware descriptions is necessary. Tools for generating C++ code that can be embedded into a Simulink environment already exist on the market. Known commercial approaches in this area are VTOC [2] and Model Studio [3]. VTOC is a model generator that allows generating C++ models from Verilog hardware descriptions. The generated models can be embedded into HDL or SystemC environments via wrappers for the Verilog PLI (Program Language Interface) and SystemC. The model generator Model Studio allows generating and validating software models. Models created by this tool can be embedded into several system level design tools as well as HDL simulators and SystemC environments. An API is generated, which provides visibility to the model's content.

Both approaches do not intend to produce user-readable code. Whereas user-readable code is required if a manual transition to a more abstract modeling of parts of the generated code is planned. These parts may be either performance-critical or their RT-level implementation is insignificant for the system model's application (ALUs, memory banks, protocol interfaces, etc.). Some non-commercial approaches generate SystemC code that allows manual adaptations [4] [5]. This class of tools is characterized by a mapping of a subset of an HDL to almost equivalent SystemC code. The original HDL hardware description is reproduced by a generated code that uses the available SystemC constructs. If there is no equivalent construct a comparable one is chosen, which is functionally equivalent in most cases. Even if there is no comparable SystemC construct, the HDL construct is not part of the supported subset. Thus, the supported subset of these tools is significantly smaller than the synthesizable subset of the HDL. An equivalent behavior of the generated SystemC model is not guaranteed due to an imprecise language mapping.

4. SystemC-based approach

Our approach for an automated transformation of hardware descriptions into Simulink models also uses SystemC as an intermediate layer. The approach can be derived in two steps. In a first step, a VHDL hardware description is converted into a functional equivalent SystemC model. This step is done using a company-internal transformation tool, which allows a highly flexible rule-based manipulation of source code descriptions. For that tool, a transformation rule was specified which provides a mapping of hardware description and SystemC language. In a second step, the resulting SystemC code is embedded into Simulink via a specific SystemC wrapper class, which implements an S-Function interface to Simulink.

4.1. Generation of SystemC models

The use of an existing transformation tool for realizing the conversion from a VHDL hardware description into a SystemC model reduced the implementation effort to the specification of a rule set for the mapping of the two languages. A powerful environment for defining this mapping is provided by a previously described in-house transformation tool [6] [7]. The main concept of the tool is a conversion of an HDL description into an XML [8] representation. The XML representation is the basis for transformations. One of the main advantages of the XML representation is that efficient standard tools can be used to manipulate the representation. These tools are XSLT processors, which use the W3C standard language XSLT [9] to define rules (so-called style sheets) for processing the input XML document. In our case, the XSLT processor Saxon [10] is used. Finally, after the transformations are performed, the XML representation can be converted back into an HDL description. The transformation tool ensures that comments and formatting is preserved from the original HDL code if the related statement is unchanged during the transformations. This allows the mixed use of manual manipulation steps (by the designer) and automated transformation steps (by the tool). Figure 2 shows the XML-based transformation flow.

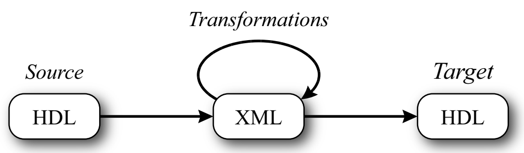

Figure 2. XML-based transformation tool

The transformation tool mainly addresses design manipulations, which are often done manually. It replaces these time-consuming and error-prone manipulations with an automated approach.

It is not mandatory that input and output languages of the tool are the same. In case of different languages, a transformation rule that describes the mapping between both languages is required. For the processing of different languages, the transformation tool requires a grammar description for each of them. As shown in Figure 3, these grammar descriptions are provided in the XML Schema format [11]. At first, the XML Schema descriptions are used to define the structure of the XML representation of a design for the particular language. The representation is based on a semantically annotated abstract syntax tree. Additionally, the tool is able to generate the environment that is necessary to manipulate code written in the particular languages that is described by the grammar descriptions. This generation process yields four documents per language. Firstly, the XML-Schema description is converted into an HTML document that describes the grammar for the user who implements transformation rules. Secondly, a grammar description for a parser generator is created. The parser generator uses this description to build a lexical scanner and a parser that are used to convert HDL descriptions into the XML representation. The transformation tool uses ANTLR [12] generated lexical scanners and parsers. Thirdly, a DTD for the XML representation is generated. The XSLT processor uses this DTD to check the syntactical correctness of the HDL code represented by the XML tree. Finally, a library of XSLT templates is created. The transformation rules use those templates to add new code to the transformation result. They ensure that the generated code is syntactically correct. The specification of transformation rules is performed in a generic XML-based transformation language that is derived from the available templates for the HDL. Rules written in this language can be automatically transformed to XSLT style sheets, which are used by the XSLT processor to manipulate the XML representation.

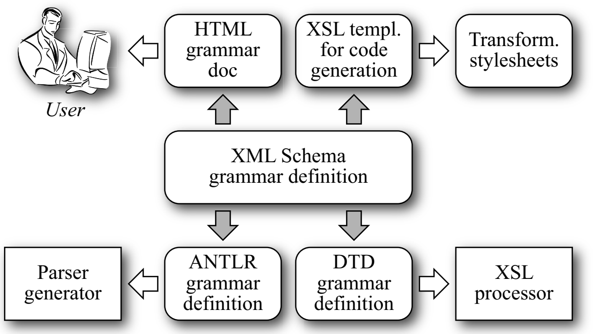

Figure 3. Automatically generated tool environment

The transformation tool includes grammar descriptions of both of the required languages: The input VHDL and the output SystemC. Therefore, the main task for the implementation of a VHDL to SystemC conversion was the description of a transformation rule that maps VHDL constructs to equivalent SystemC.

Because SystemC does not provide types that behave exactly like VHDL types, a mayor requirement for the mapping was the implementation of equivalents for the VHDL standard data types (i.e. integer, arrays, and enumeration types). The implemented mapping bases on the use of a library that contains classes and templates to reproduce the VHDL standard types including their associated attributes and operators. VHDL types derived from the standard types can then be handled by the VHDL-to-SystemC mapping. To allow compatibility to standard SystemC designs, an automatically generated wrapper on top of the design hierarchy realizes a configurable conversion to standard SystemC types.

4.1.1. Conversion strategy

At first sight, RTL descriptions in SystemC and VHDL look very similar. However, unfortunately there are some differences in the details. In both languages exist hierarchical modules and processes, ports and signals, and RT level types like logic vectors and arbitrary integer types. Nevertheless, some features known from VHDL are not supported or behave different in SystemC. The differences can be classified into three categories:

1. Syntactical differences. E.g., the declaration of a function in another function is allowed in VHDL but not in C++ and therewith not in SystemC. Furthermore, C++ supports only constant integer expressions in case-statements, in VHDL array types can be used as well.
2. Behavioral differences. E.g., operator precedence is different in both languages and in contrast to VHDL, SystemC does no boundary checks on variable assignments.
3. Language elements. E.g., multiple architectures for one entity, subtypes, or attributes of objects and types.

The main task for a conversion from VHDL to SystemC is now to find equivalent C++-constructs for VHDL elements. Solving the problems of the first category is possible, but only by workarounds because the formal syntax of C++ cannot be touched. Functions can be declared in a particular namespace instead of in another function and accessed with its explicit name; case-statements can be replaced by if-elsif-statements when necessary. The second category splits into two parts. Some of the problems, like operator precedence, are actually caused by the C++ syntax definition. Workarounds are the only possible solution here as well (e.g., brackets can be used to correct operator precedence). The boundary checks on variable assignments arise from the differences in the type systems of C++/SystemC and VHDL. Those differences can be eliminated by implementing the missing features in C++. The third category, VHDL language elements that are not available in SystemC, can also be solved by implementing counterparts in C++. E.g., multiple architectures for one entity can be realized as different derivations of the same entity module. Many of the disparities apply to types and objects of those types. Therefore, it is useful to provide a set of types and associated operators that behave like those in VHDL. Furthermore, they should support subtypes and the VHDL attributes. Because IP blocks should be converted into SystemC and, in general, such blocks solely contain synthesizable code, it is sufficient to consider the synthesizable types (i.e. integer, enumeration, and array types) in this case.

In our approach, a library has been created that contains C++ template classes. They can be used to declare VHDL-like types. Furthermore, it contains declarations of the VHDL standard types as declared in package std.standard and macros for the declaration of user-defined types.

The provided types and objects of those types support the following features known from VHDL:

- attributes
- range and index constraints
- boundary checks on assignments
- ascending and descending ranges that do not have to start with 0
- constrained and unconstrained subtypes
- operators as defined in VHDL
- string-to-array conversion.

Not implemented until now is the support for aggregates and port slicing.

Using these types instead of natural C++ or SystemC types allows a straightforward conversion of type and object declarations, computational and assignment statements, and expressions in VHDL to equivalent SystemC. The resulting code is still compliant to standard SystemC.

4.1.2. Implementation of the transformation rule

The VHDL to SystemC transformation works as described in section 4.1. The transformation rule walks through the XML representation of the VHDL code that has been set up by the front-end of the transformation tool. While walking through the tree, every VHDL element is replaced by an equivalent SystemC construct and afterwards the resulting XML representation of SystemC code can be written out as source text.

Table 1. Mapping VHDL to SystemC (extract)

	VHDL	SystemC
design units	package	namespace
	entity	SC_MODULE
	architecture	SC_MODULE derived from entity module
declarations	generic	template parameter
	port	sc_port
	signal	sc_signal
statements	process statement	SC_METHOD/SC_THREAD
	concurrent statement	equivalent process statement

In a first step, the transformation rule transforms package declarations to namespaces and the declarations inside of it to equivalent C++ declarations. In a second step, it processes entities. They lead to SystemC modules, its generics to template parameters, and its ports to corresponding sc_port declarations. To support multiple architectures for one entity, the transformation creates for each architecture one module that is derived from the module that replaces the entity. Then, the content of the particular architecture is processed sequentially. Signal declarations lead to declarations of sc_signal, component instantiations to instantiations of modules as class members, process statements to SystemC processes, and so on. SystemC does not support concurrent statements as known from VHDL but for each kind of it, there is an equivalent process statement. Finally, a constructor has to be generated that binds the ports of embedded components and starts the processes.

For the design's top module, an additional step is performed. To connect the design to a standard SystemC environment, the top module is surrounded by a wrapper that converts its ports into standard SystemC types. Here it is possible to use a standard mapping for the type conversion, or it can be configured individually.

A particular characteristic of this transformation is that it is implemented to create readable code. As far as possible, identifiers and the code structure in the generated code are the same as in the original code. This especially includes formatting characters and comments; they are inserted at a position in the result code that is equivalent to their original position. Hence, the design is prepared for further processing steps, automatic or by hand.

4.2. Embedding SystemC into Simulink

The integration of the resulting SystemC code into Simulink is done by a previously described wrapper concept [13]. This concept bases on an abstract SystemC wrapper class that realizes the S-Function interface required by Simulink. The wrapper class provides the necessary functionality for synchronization and data type conversion between SystemC and Simulink. Classes implementing the abstract SystemC wrapper class are generated along with the VHDL-to-SystemC transformation. Therefore, the user of the flow described above gets a ready-to-compile Simulink model of the VHDL design.

The synchronization of the Simulink and the SystemC environment is a complex task because Simulink uses a sample-time whereas SystemC uses an event queue. Integrating SystemC into Simulink as an S-function requires that the S-function is updated whenever a signal at the interface of the SystemC module changes. Therefore, a synchronization method where incoming signals inherit their sample time and outgoing signals have a variable sample time would be preferred. Due to limitations of Simulink, this preferred method is not supported. As a result, different scenarios with different synchronization strategies are supported by the provided wrapper classes. Table 2 gives a short overview.

Table 2. Synchronization strategies

Scenario	Strategy
SystemC module has no incoming signals	Only internal events can occur. The S-function gets a variable sample time assigned. It uses the event queue to predict the time of the next event.
SystemC module has incoming signals but no internal periods	Events can occur if incoming signals change. The S-function gets inherited sample times assigned. The sample time is inherited for each port separately in order to allow different input sample times.
SystemC module has incoming signals and internal periods	The S-function inputs inherit their sample times. A base period is calculated from the inherited sample times and the internal periods. The base period is assigned to the outgoing signals.

The data type conversion is done implicit by the wrapper classes. Default C/C++ types are mapped to their Simulink equivalents. For SystemC types, different mapping strategies are available. The simplest way is a conversion to the Simulink standard type double. For typical SystemC modules generated from VHDL a conversion from SystemC logic vectors to Simulink fixed point is the appropriate way.

5. Experiments

In the following, the automated flow will be compared to a manual. An automotive electronics example, the implementation of a configurable bus-arbiter, has been used for the evaluation. As shown in Figure 4, the arbiter sequences the requests of the masters to access one of the slaves. The master and slave modules are part of the system model, which is implemented in MATLAB/Simulink. The arbiter itself is available as an IP block implemented in synthesizable VHDL.

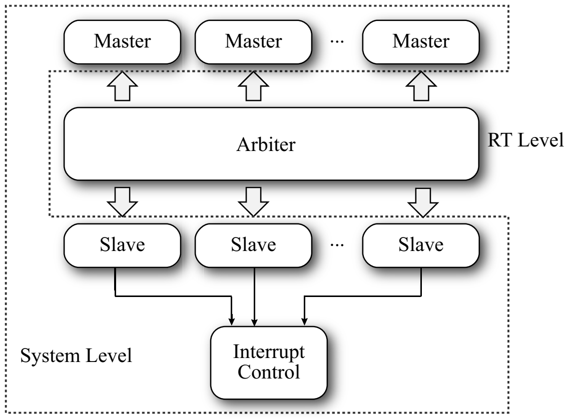

Figure 4. Bus-arbiter model

For the manual conversion of the arbiter to a Simulink component as described in section 2, the designer needs to know its entire functionality and how he can express it in C code. Additionally, he must know how to write an S-Function. Due to the different models computation in VHDL and Simulink simulations, it is a difficult task to come to a comparable behavior. As one can see, a designer with knowledge in several abstraction layers and their particular tools is necessary to manage this task, and even then, it is time-consuming and error-prone.

The automated flow consists of only four commands to the transformation tool, load bus-arbiter design, transform to SystemC, generate S-Function wrapper, and write out design. This can be done either by the system engineer or by the RTL engineer because the four commands automatically create an S-Function and no special knowledge is needed. For both of them, the complete conversion costs only about five minutes.

In both approaches, the final step is to compile the resulting S-Function and instantiate it in the appropriate Simulink block, and afterwards, the simulation of the system can be started.

The analysis of the simulation performance shows that it is almost the same in the two cases. Table 3 lists some profiling results that have been achieved with MATLAB R2008a running on a Linux system with an Intel Pentium D 2.8GHz CPU with 2GB RAM. The mentioned values are the average values of ten simulation runs in each case.

Table 3. Simulation Performance

	manually implemented	automatically generated
total duration	43.23s	44.03s
arbiter output	0.68s (1.57%)	1.72s (3.91%)
arbiter update	0.63s (1.46%)	-
arbiter output + update	1.31s (3.03%)	1.72s (3.91%)
arbiter relative performance	1	0.76
total relative performance	1	0.98

The total duration of the simulation is 43.23s with the manually implemented S-Function and 44.03s with the automatically generated one. The Simulink simulation kernel calls two methods of the bus-arbiter S-Function in each time step, the output method to set the modules output ports and the update method to update the modules internal states. In case of the manually implemented S-Function, the simulation spends 1.31s on these two methods; that corresponds to 3.03% of the total simulation duration. The automatically generated S-Function stores and updates its internal state in the SystemC part. Therefore, the update method is not used here. The output method consumes 3.91% of the total simulation duration, 1.72s. The simulation performance of the automatically generated arbiter module considered separately decreases by a factor of 0.76 compared to the manual implemented version. That is very little according to the fact that it can be generated in a few minutes. The performance of the complete simulation merely decreases by a factor of 0.98. If one assumes that the proposed flow is only used for a few parts of the system and most of the simulation duration is spent by processing the rest of the system, one can conclude that the automated flow involves no significant overhead.

6. Conclusion and future work

In this contribution, the challenges that lie in the integration of hardware IP into the system development flow have been discussed. Furthermore, an automated flow has been described that allows the seamless integration of IP components available as VHDL code into a Simulink model. The VHDL code is converted automatically into equivalent SystemC code. This is done by an environment for automated design transformations that has been explained in detail as well as the VHDL-to-SystemC mapping strategy. The result is a SystemC module surrounded by an S-Function wrapper, which can be used directly in a Simulink model. An analysis of the benefits and the overhead of the automated flow has been done by means of an industrial automotive electronics example. It has shown that the time saving due to the automation of the conversion surpass the overhead during simulation by far.

Further tasks are to provide support for additional VHDL-features and to add specialized implementations of often-used VHDL types in C++ to improve their performance.

7. References

[1] Hoffman, M., T. Beaumont, Application Development: Managing a Project's Life Cycle, *Mc Press*, 1997

[2] ARC International, VTOC for SoC Models, http://www.arc.com/

[3] Carbon Design Systems, Model Studio, http://carbondesignsystems.com/

[4] VHDL-to-SystemC-Converter, European SystemC Users Group, http://www-ti.informatik.uni-tuebingen.de/~systemc/

[5] VH2SC, HT-Lab, http://www.ht-lab.com/

[6] Oetjens, J.-H., J. Gerlach, W. Rosenstiel, An XML Based Approach for the Flexible Representation and Transformation of System Descriptions, *Forum on Design Languages*, Lille, 2004

[7] Oetjens, J.-H., J. Gerlach, W. Rosenstiel, Flexible Specification and Application of Rule-based Transformations in an Automotive Design Flow, *Design, Automation, and Test in Europe*, Munich, 2006

[8] World Wide Web Consortium, Extensible Markup Language, http://www.w3.org/XML/

[9] World Wide Web Consortium, XSL Transformations (XSLT) Version 1.0, http://www.w3.org/TR/xslt/

[10] The SAXON XSLT and XQuery Processor, http://saxon.sourceforge.net/

[11] World Wide Web Consortium, XML Schema 1.0, http://www.w3.org/XML/Schema/

[12] ANTLR, ANother Tool for Language Recognition, http://www.antlr.org/

[13] Hylla, K., J.-H. Oetjens, W. Nebel, Using SystemC for an Extended MATLAB/Simulink verification flow, *Forum on Design Languages*, Stuttgart, 2008