Flexible Specification and Application of Rule-based Transformations in an Automotive Design Flow

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

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

Wolfgang Rosenstiel
Universität Tübingen
Wilhelm-Schickard-Institut
Sand 13
72076 Tübingen
rosenstiel@informatik.uni-tuebingen.de

Abstract

This paper addresses an XML-based design environment, which provides a powerful basis for the manipulation of hardware design descriptions. The contribution of the paper is a flexible specification entry for the definition of transformation rules, which allows a designer to specify transformations by his/her own without having XML expertise. The specification entry provides a guided and graphically supported mechanism to define transformation rules. This opens up a new approach, in which the specification and verification of a transformation rule is carried out by using simple design examples, to be applied to arbitrary complex designs subsequently. A new key characteristic of our approach is that both transformation environment and transformation entry tool are based on a very compact definition of the hardware description language grammar in use, and both of them are fully automatically generated from that basic grammar definition. This makes our approach highly open for other hardware and system specification languages. The paper describes the transformation environment and transformation entry tool, and demonstrates its application in terms of two automotive-typical transformations, addressing power aspects on the one hand, and safety aspects on the other.

1. Introduction

The increasing gap between design complexity and productivity demands a permanent improvement of the design process. To meet this challenge, an extensive automation is required that limits manual interventions during the design process to the vital points. Major parts of these manual interventions are design transformations. In an automotive design flow, these design transformations range from simple design adaptations to complex optimizations. Typical examples are safety-enhancing transformations (e.g. adding bus redundancy) or power optimizations for power-critical applications (e.g. central locking remote controls). Automating these transformations replaces the local view of a designer, which often limits optimizations to a restricted area of the design, by a global view that allows to utilize a larger optimization potential. In addition, performing transformations manually is an error-prone process. Finally, in an industrial environment those time-consuming manual transformations are often skipped due to time-to-market restrictions, which results in suboptimal design implementations.

Many existing EDA tools integrate automated transformations for design optimization. The wide range of requirements in design processes often exceeds the set of transformations, which these tools support. A flexible methodology, which allows user-defined transformation rules, addresses these requirements. It combines the flexibility of manual transformations with the efficiency and reliability of automated transformations.

In [1] we introduced a flexible environment for the transformation of hardware designs. The proposed environment uses the metalanguage XML [2] for design representation. We showed that the use of XML allows an efficient handling and transformation of design data. The transformations are performed by widely used XML tools, which initially require the definition of a transformation rule. This rule definition is a complex process that requires a lot of XML expertise. In this paper, we present a transformation entry tool that offers a view on these transformation rules, which is common for a hardware designer. According to this process, rules can be defined within a fraction of the time compared to a manual implementation, and there is no need of XML expertise.

The paper is organized as follows: In section 2, related work dealing with design transformations and relevant XML-based approaches in the EDA context are discussed. Section 3 presents our transformation environment and its new structure based on a compact grammar definition. In section 4, the transformation entry tool is

3-9810801-0-6/DATE06 © 2006 EDAA

described. This tool was used to specify the transformations, which are presented in section 5. Finally, section 6 concludes with an outlook on future work.

2. Related work

Today, various commercial tools transform design representations. Most of these tools have a closed architecture preventing an adaptation to individual requirements. A typical example of this are synthesis tools, which usually support a restricted but fixed set of transformations. With several optimization steps, they transform an internal design representation, but the hardware designer is restricted in adding individual optimizations. Therefore, the designer has to implement his/her idea manually or has to discuss an adaptation with the vendor of the tool.

A first step in increasing the customizability was done by code analyzing tools like SpyGlass [3]. This RTL analyzer is able to perform code checks with specified rules. A graphical user interface allows the customization of these rules and Perl functions allow the specification of user defined rules. These Perl functions are implemented in a C API that provides direct access to the internal design representation. Although SpyGlass has a tool-specific data structure this approach allows a high degree of customizability.

When looking at approaches that allow direct access to their data structure, XML and OpenAccess [4] play an important role. While XML is a language-based approach, OpenAccess represents an open standard for a design database and data API. The C++ API of OpenAccess is optimized for efficient access to the design data for EDA tools, but in contrast to XML, there are no standards for transformations available. Even though these standards are not used by the majority of the described XML-based approaches in the EDA context [5][6][7], an application using XML transformations is published in [8]. Following the example of Javadoc [9], this approach automatically generates documentation of VHDL code.

3. XML-based design transformation

Our approach replaces manual design transformations by a methodology that is based on the use of XML for design representation and transformations. XML, which is well known and widely used in the area of internet applications, provides features to create a flexible basis for an expandable transformation environment. Many standard tools offer XML interfaces and there are several languages for the transformation of XML documents available. The most popular of these languages is XSLT [10], which is also the most important part of the XSL standard. This language is used to implement XSL stylesheets that describe the transformation of XML data. The resulting document can be either XML or any other non-XML format. Transforming the contents of XML documents allows to process an XML design representation. In our approach, we use XSL to describe design transformations, which enables to take advantage of existing tools to perform transformations. These XSL processors compile stylesheets into programs, which are highly optimized to work on XML data. This results in good performance of common XSL processors. Compared with dedicated in-house software solutions, we benefit from existing and technically mature tools and save development time.

3.1. XML data structure

Even though XML is mainly used in the web context, its features support a wide range of other applications. Because XML allows an almost unrestricted document structure definition, it is possible to define an XML dialect adapted to a specific application. In our transformation environment, the HDL parser directly generates this XML structure by mapping its syntax tree to XML. Beyond this syntactical information, it is partially necessary to support design transformations by adding semantic information to the XML document. Instead of an extraction of this information by every single transformation, an annotation of the XML document makes reuse of the semantic information possible. An efficient method to store this information in an XML structure is the annotation of the XML tree with XML attributes. In this manner, the information is stored in the nodes and no edges or nodes have to be added to the tree.

The definition of this data structure is realized in XML Schema documents [11]. XML Schema is an XML language and can be used for an automated check of the design representations for syntactical correctness.

3.2. HDL-specific data

We use a grammar definition in XML Schema not only for checking XML design representations but also as basis for the transformation environment. By an automated generation of all grammar-specific components, the support of a HDL just requires an XML Schema representation of the appropriate grammar. With this novel XML-based tool implementation, an easy adaptation of our approach to the requirements of different design flows is possible. Figure 1 shows an overview of the documents derived from the XML Schema definition of the grammar.

When an XML Schema description of an HDL grammar is available, a set of XSL stylesheets is used to generate the derived documents automatically. First, the XML Schema formatted grammar is translated into a user-readable HTML format, which provides a reference for the HDL grammar. Second, a library of XSL templates for code generation is automatically generated. Transformation stylesheets use these templates to replace

existing HDL code, which is shown in a detailed description of the transformation process in section 3.4. Third, a DTD compliant subset of the XML Schema definition is translated into the DTD format. This allows the use of non-Schema aware XSL processors. Finally, the XML Schema file is transformed into a grammar description for the parser generator ANTLR [12]. This description in a tool-specific format is used by ANTLR to generate the parser for our transformation environment.

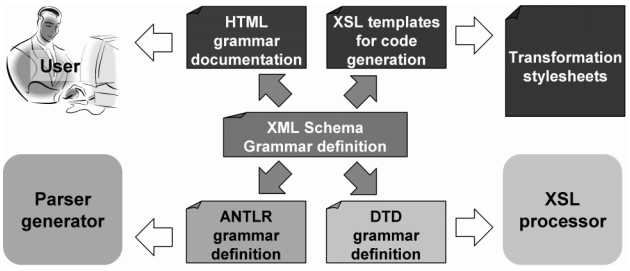

Figure 1. Documents derived from the grammar

3.3. Transformation environment

The parser is the starting point of our transformation environment and represents an interface to the existing design flow. The Java-based analyzer for HDL hardware descriptions generates XML design representations, which are handled by an XSL processor for further processing. The use of Java allows the integration of Saxon [13], which is a common XSL processor. Furthermore, the platform independence of Java is useful for integrating the transformation environment into different environments. Figure 2 shows the processing steps of the transformation environment.

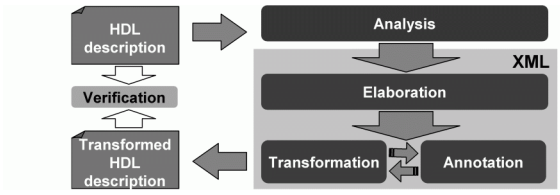

Figure 2. Transformation environment

After analyzing the HDL description, an elaboration step is required. In this step, the analyzed components of the design are stored in a common XML design representation. In our methodology, this and subsequent steps are implemented by XSL stylesheets. The elaborated design provides the basis for the transformation and annotation steps. The annotations are used to setup the XML design representation for the transformations by adding semantic information. The amount of annotations depends on the intended design transformation. The design transformations can be partitioned into smaller transformation steps executed sequentially. This allows us to build complex transformations in a bottom-up manner based on a set of basic transformations steps. The required sequence of annotations and transformations is checked before a transformation is started. If a transformation causes invalid annotation information, the annotation step is removed from a transformation history list, which is stored in the XML tree. If this annotation information is needed in a following transformation step, the annotation step has to be repeated. The correctness of those transformation steps, which shall generate functional equivalent code, is proved by an equivalence check. This is performed by a commercial formal verification tool that compares given HDL descriptions. These descriptions can be exported from the XML representation after each processing step.

3.4. Design transformations

The transformation step described in the last section is realized by XSL stylesheets, whose implementation follows a basic structure. This structure is adapted to the specific transformation task. Figure 3 shows the flow of XSL design transformations.

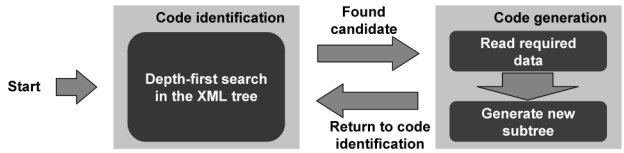

Figure 3. Flow of design transformations

A first partitioning can be made by differentiating between code analysis and code generation. While the code analysis phase identifies the subtrees of the XML tree that are relevant for a given transformation step, the code generation phase replaces those subtrees by new ones. The code analysis performs a depth-first search in the XML tree. If a subtree is a candidate for a transformation, the code generation starts. The first step of the code generation is an extraction of the required data. This data is used to generate a new subtree before the code analysis of the next element is performed.

The described transformation environment is used in an automotive design flow. It is available in three versions. An applet and an application version provide a graphical user interface. Additionally, a command line version allows to use the transformation environment in batch mode.

4. User-defined transformation rules

Taking advantage of the full flexibility of the XML-based transformation environment users can define their own transformations. These XSL stylesheets are a very efficient way to describe transformations on XML documents. A single XSLT statement is able to substitute many lines of code in conventional languages. Even so, the size of a complex transformation rule can easily exceed 1000 lines of XSL code. To support the user in implementing or adapting a transformation rule we have

defined a systematical process for the transformation rule entry. Figure 4 illustrates this process.

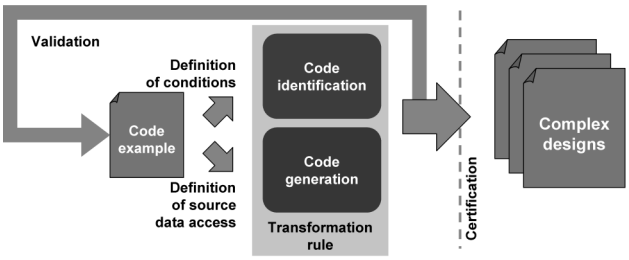

Figure 4. Specification of transformation rules

The entry process is based on a code example that represents the starting point of the transformation task. This corresponds to a designer's "natural way of thinking", which is example-based rather than rule-based. With a short code example, covering the aspects that are relevant for the addressed transformation, the designer is able to define the transformation rule without having the overhead of non-relevant code. This example is used to define conditions for the transformations and the required data of the source design. In a continuous validation process, the user compares the transformation result with the specification. If the specification is met and the transformation shall be function preserving, a formal verification may be used to prove the functional equivalence of the source and the target design. After a successful validation of the rule, it can be applied to arbitrary complex designs.

The described process for the transformation rule entry is implemented with a graphical tool that guides the user through the required steps and automatically generates an XSL stylesheet. In a sequence of dialogs, the tool interacts with the user to retrieve the required data. This data can be defined manually or by graphically selecting corresponding segments it in the HDL code of the code example. Figure 5 shows the main steps of a rule specification.

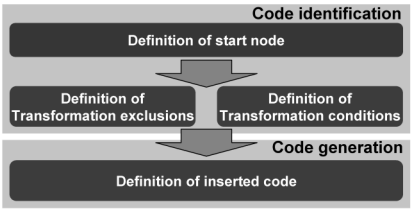

Figure 5. Transformation entry tool

Using a reset transformation for VHDL processes (asynchronous reset ↔ synchronous reset) as an example for the specification of a transformation rule, the transformation rule entry begins with the definition of a process statement as the start node. This start node defines the transformed functional unit of the source design. After the start node has been selected, exclusions and conditions for the transformations may be defined. The exclusions are used to specify segments of the source code not to be treated by the transformation step. For the reset transformation, VHDL process statements without a clock are excluded. The next step, the definition of transformation conditions, allows to perform different transformations for different segments of the source code. In case of the reset transformation a differentiation between process statements with synchronous and asynchronous resets is needed to allow a bidirectional transformation. During the code generation, the transformation entry tool guides the user through the code definition by offering exclusively the syntactical correct statements. When data from the source design has to be copied into the transformation's target, a dialog allows to define this data by graphically selecting it in the HDL description of the code example. The transformation entry tool retrieves the HDL-specific information, which is required for the code generation, from the existing XML Schema description. In the same way as the transformation environment does, the transformation entry tool only requires this XML Schema grammar definition to adapt the language-specific tool components to other HDLs.

The transformation entry tool allows an intuitive definition of transformation rules. This is achieved by input dialogs that guide the user through the rule definition process. The entry tool was used to define the reset transformation rule, used as an example before, as well as numerous other rules. In contrast to hand coding in XSL, the definition of a rule needs a fraction of the development time and is less error-prone.

5. Experiments

In this section, we present a range of design transformation rules to demonstrate the flexibility of our approach. In the current state of our transformation environment, the rule set covers several design manipulations ranging from simple code adaptations to complex optimization steps. The currently available transformation rules are shown in Table 1.

Table 1. Available Transformation Rules

Category	Transformation rule
Support of reusability	Separation of structural and functional code
	Statement transformation: Concurrent signal assignment ↔ combinatorial process
	Reset transformation: Asynchronous ↔ synchronous
	Code scrambler
Optimization of synthesis and verification results	Power optimization by clock gate insertion
	Enhancement of the significance of line coverage analysis by if-statement separation
Automation of manual transformation steps	Completion of sensitivity lists
	Identification and removal of dead code
	Insertion of error correction for busses

The transformation rules can be classified into three categories, including rules for improving the reusability characteristics of a design (category one), for the optimization of synthesis and verification results (category two),

and for the automation of typical manual transformation steps (category three).

The following sections will show two examples for transformation rules in detail: The first rule, a clock gating transformation, describes an optimization transformation. The second rule addresses the safety requirements of the automotive domain by automating the manual insertion of an error correction mechanism in bus-based architectures.

5.1. Clock gating transformation

Today the power dissipation of integrated circuits becomes more and more important. Typical examples for low power applications in the automotive domain are central locking remote controls. Important parameters for the power dissipation in integrated circuits are the registers. This is addressed by inserting a clock gate into the clock path of the registers. Common clock gating approaches use either manual clock gating on major design components or an automated insertion of clock gates on gate level. The manual approach benefits from the designer‘s knowledge of the design functionality but it is error-prone, time-consuming and in many cases limited by the restricted view of the designer. Whereas an automated clock gate insertion on gate level provides good optimization results, but prevents further adaptation by the designer, who uses RTL for specification. In our approach, the transformation works on RTL, which means on the same abstraction level as the designer during the specification. This allows to process functional units, which the designer intuitively combines. On the gate level, information about these functional units would be lost. In addition, because the transformation result is provided in the designer’s typical HDL environment, further adaptations are possible. The basic idea of the transformation is to analyze the RTL description to insert clock gates into the clock path of registers with potentially inactive inputs. The clock gate is controlled by a combinatorial logic, which is automatically determined. Figure 6 illustrates this process.

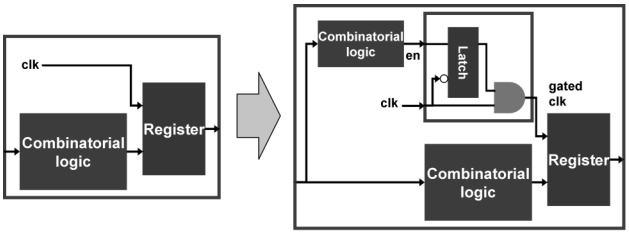

Figure 6. Clock gating transformation

The clock gating transformation is partitioned into three processing steps. These steps are performed by a set of smaller transformations. The transformation starts by selecting sequential VHDL processes that are suited for clock gating. Therefore, a static analysis checks each process for potentially inactive phases. After the processes have been partitioned, a second transformation step optimizes the conditions of if- and case-statements in processes allowing clock gating. The transformation finally inserts the clock gates with the appropriate control logic. Table 2 (second column) summarizes the required time for conception, implementation, and application of the transformation rule on an automotive GPS correlator design, which is part of a car navigation system.

Table 2. Transformation rules

	Clock gating transformation		Bus transformation	
Number of XSL stylesheets	3		1	
Lines of XSL code	4462		1856	
Conception of transformation process	10 days		1 day	
Rule specification with the entry tool	3 d.	-	0.5 d.	-
Hand coding of the rule (estimated)	-	15 d.	-	5 d.
∑ with / without entry tool	**13 d.**	**25 d.**	**1.5 d.**	**6 d.**
Transformation on a Sun Blade 2500	149 seconds		47 seconds	
Manual transformation (estimated)	10 days		2 days	

It is shown that, compared with a hand coding of the transformation stylesheets, the rule specification with our entry tool requires a fraction of the time. Through this, a summation of conception and implementation time is only slightly longer than the required time for a manual transformation. This additional time is more than compensated by the reusability of the rule for future designs. A comparison of our RTL power optimization transformation with a commercial tool working on gate level is shown in Figure 7.

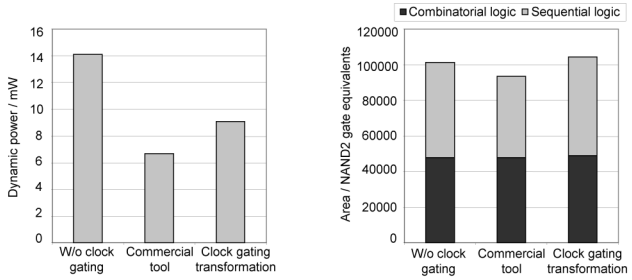

Figure 7. Results of the clock gating transformation

Due to its technology-specific optimization, the tool is able to provide slightly better dynamic power results. Comparing area, the transformation result is quite similar to the original design, while the commercial tool reaches a reduction of sequential logic. This is achieved by removing those feedback multiplexers in sequential logic cells that the transformation rule keeps to allow the verification with the integrated verification tool. The experiments show that this flexible RTL optimization transformation, which was realized in 13 days, can reach similar results compared to less flexible gate level approaches.

5.2. Bus transformation

Beyond the behavior-preserving transformations described above, an automotive design flow has special re-

quirements that exceed typical optimization transformations. One of these typical requirements, which our flexible approach addresses, is safety. In the following section, we show a transformation implementing an automated insertion of error detection and correction in bus-based architectures. A widely used approach for redundant data transfer is the Hamming code [14], which is able to correct one-bit errors and detect two-bit errors. In the following, this coding scheme is used as an example and other redundancy mechanisms may be realized in a similar way. The common way of realizing encoded busses is the time-consuming manual insertion of encoder and decoder blocks. If no parameterizable encoder and decoder block is available, a specific solution for the current design has to be implemented, which is rarely reused in other designs. In our transformation, we automatically insert parameterizable encoder and decoder blocks. As shown in Figure 8, this is implemented by inserting a wrapper around the components with bus interfaces and attaching encoder and decoder blocks to the interfaces.

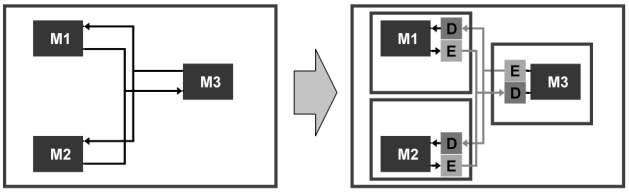

Figure 8. Bus transformation

As the transformation described in the last section, the insertion process is performed in three steps. The first step is the determination of components attached to the considered bus. After that, these components are encapsulated into wrappers and encoder and decoder blocks are inserted at the interfaces. The parameterizable encoder and decoder blocks are configured in the last step, where the required bus width is determined and the bus is adapted. The required time for conception, implementation, and application of the transformation rule on an example design is summarized in Table 2 (third column). The design contains a bus arbiter and several master and slave modules, which are connected with data and address busses. In this example, the required time for a manual bus transformation already exceeds the time for the conception and the entry tool based specification of the transformation rule.

Both transformation examples increase the design complexity and show that design transformations beyond typical optimizations play an important role in automotive design. This underlines the importance of the flexibility of our XML-based transformation approach.

6. Conclusion and future work

In this paper, we have presented a flexible specification entry for the definition of XML-based transformation rules. The entry provides a guided and graphically supported mechanism for the definition of transformation rules. This opens up a new approach, which corresponds to the designer's "natural way of thinking". Specification and verification of a transformation rule is carried out by using simple design examples to be applied to arbitrary complex designs subsequently. As shown in the experiments, this allows an efficient specification of transformation rules and helps to replace time-consuming and error-prone manual transformation steps. The implementation of the transformation entry tool and the transformation environment is based on a very compact definition of the underlying HDL grammar. This new feature of our approach is realized by automatically generating language-specific tool components from the basic grammar definition. It opens our approach to other hardware and system specification languages. In combination with the simple specification of transformation rules, this approach could be easily extended to other application domains.

Future work will focus on supporting further hardware and system specification languages as well as transformations between different languages.

7. References

[1] 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

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

[3] Atrenta Inc., SpyGlass Predictive Analyzer, http://www.atrenta.com/

[4] Si2, OpenAccess Project, http://www.si2.org/openaccess/index.html

[5] Lee, E. A., S. Neuendorffer, MoML A Modeling Markup Language in XML, *International Conference on Computer Aided Design*, San Jose, 2000.

[6] Reshadi, M. H., B. Gorji-Ara, Z. Navabi, HDML: Compiled VHDL in XML, *VHDL International Users Forum Fall Workshop*, Orlando, 2000

[7] Zamfirescu, A., Z. Zhao, HXML – A New Approach to Managing Hardware Information, *Forum on Design Languages*, Lyon, 1999

[8] Ecker, W., M. Heuchling, J. Mades, T. Schneider, A. Windisch, K. Yang, Using XML for VHDL Model Representation, *World Computer Congress*, Beijing, 2000

[9] Sun Microsystems, Javadoc Tool Home Page, http://java.sun.com/j2se/javadoc/

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

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

[14] Hamming, R. W., Error Detecting and Error Correcting Codes, *Bell System Technical Journal.* 29: 147, 1950