

Quality Guaranty of Reusable Circuit Descriptions through HDL Source Code Analysis

Steffen Rülke¹⁾, Jörg Schneider¹⁾, Frank Rogin²⁾, Peter Bachmann²⁾

¹⁾ FhG IIS, EAS Dresden, Zeunerstr. 38, D-01069 Dresden {ruelke|jsch}@eas.iis.fhg.de

²⁾ Cottbus University of Technology, P.O. Box 10 13 44, D-03013 Cottbus, {fr|pb}@informatik.tu-cottbus.de

Abstract

The standardized formal specification of microelectronic circuits is a precondition for computer-aided design reuse. Unfortunately, the hardware-description languages (HDL in short) that are usually used for the formal specification offer a variety of alternative means. This may cause serious problems in the communication between human designers and, particularly, within the reuse process. Guidelines to restrict the freedom in HDL coding enforce a standardized coding style. Consequently, analyzers for checking HDL code against a given set of coding style guidelines are important to achieve a high design quality in a reuse-based design flow. The present paper describes an approach for the efficient analyzer generation regarding a given set of guidelines. It applies proved strategies known from compiler generator techniques and utilizes a C/C++ user interface.

1 Introduction

1.1 Motivation

Because of the steady growing integration density in microelectronics the complexity of integrated circuits is redoubled every 1.5 years [1]. Further significant challenges for design strategies are caused by time-to-market requirements which become more and more the most important design constraint. Computer-aided reuse of already designed components (also called IP - Intellectual Property) is the key to cope with the available high design complexity in the required short design time.

Hardware description languages are proven specification means for microelectronic systems. In practice, standardized HDLs like VHDL or Verilog are also applied to specify IPs or to describe designs for IP integration. Such HDLs enable the CAD-conform formal specification of design objects for synthesis, validation, or retrieval at different levels of abstraction (e.g. behavior, RT, logic gate). Moreover, these HDLs provide flexible and adaptable generic concepts for the parametrization of circuit descriptions. Furthermore, HDL specifications can be easily modified. If a predestinated language standard is kept, HDL porting into a different design environment is possible, too.

However, serious problems of HDLs are the variety in alternative specification means, the diversity in phraseology, and the inherent ambiguity or redundancy. A couple of different specification ways for the same functionality yield to particular personal coding styles of the human designers. This worsens the adaptability of an IP or circuit description into another design environment with a distinct design flow and different design tools. Further problems are the more complicated communication between designers, the rejection of particular circuit description by design tools if they only accept a HDL subset, and difficulties in IP retrieval searching databanks.

To reduce the prementioned problems and therefore, to achieve a higher design quality, circuit descriptions by HDLs should meet a certain standard defined by coding style guidelines. Coding style guidelines always restrict the use of any HDL. Following such guidelines, HDL code can be adjusted for particular requirements: e.g. for optimal synthesis results using a concrete synthesizer, to fasten circuit validation using a given simulator, or for simplified identification of the HDL code in a data bank. Besides, coding style guidelines contribute to create an efficient IP code since flexibility and portability of reusable components are affected by the way of coding, too.

In practice, coding standards are usually derived by requirements of the application class, e.g. IEEE 1076.6 for synthesis-suitable descriptions. Coding standards for reuse-conform specifications are provided, e.g. in Reuse Methodology Manuals [6], [9]. Other guidelines confirm tool- and technology-specific standards or recommendations, e.g. for HDL descriptions fitting Actel target technology [8]. Besides, a couple of firms have enforced house-internal standards.

To attest whether or not a given HDL specification observes a certain coding style guideline, suitable analysis tools are necessary. The implementation of coding style guidelines in the analysis tool should be facilitated by an efficient, flexible, and user-friendly strategy applying extendable and portable software technologies.

1.2 Requirements

In order to check by a tool whether the restrictions of coding guidelines are observed it is very important to distin-

guish the arts of guideline rules with respect to the different levels in the structure of a HDL. So, for instance, the guidelines may concern:

- the typographical level: formatting like indents, tabs, newlines, line length, ...
- the lexicographical level: capitalization, length of identifiers, use of special comments,...
- the syntactical level: exclusion of special kind of statements, nesting of statements, combination of statements,...
- the semantic level: ranges of types, cardinalities, relations between parts of descriptions,...
- the level of the environment: structure and organization of file-system, special interpretation of code by context-information.

The different restrictions may be formulated by the notions of the corresponding level alone, e.g. forbidden tabs as pure typographic restrictions, or they depend on several levels like context-sensitive identifiers, indents in front of nested statements a.s.o., and therefore, have to be expressed by mixed notions.

By "instance of a HDL" we denote the code, i.e. the text, for the specification of hardware (circuit). The semantics of an HDL and therefore the behavior of hardware described by such an instance, is either formally defined or - at least - by the output of a simulator. Such a simulator may be considered as an interpreter of an HDL which evaluates imperative parts of the instance - like set operations, arithmetic operations, repetitions, switches, logical operations, conditions - as well as declarative parts - like type definitions, register descriptions.

It is also important that much of the information which determines the behavior of the described hardware must be derived from the environment in which the instance is embedded. That means, beside the analysis of syntax (scanning and parsing) we may need additional knowledge stored in the environment. Moreover, the kind of this knowledge depends in general on the instance itself. As an example, the simulator, if any, must refer to a certain library in order to identify the function of a component specified by name and ports. Analogously, special comments may refer to a file where the meanings of them are defined.

2 Analysis concepts

2.1 Classification of methods

In order to implement the coding style guidelines, we can implement an analyzer using various approaches. The particular approaches mainly differ in the formal way how the guidelines become known for the analysis software. The common characteristic of all approaches is that they read in an instance of HDL and, if required, additional information from the environment regarding the context specific seman-

tics. As output, each analyzer produces messages about place (e.g. line and column in the specification) and type of guideline violation.

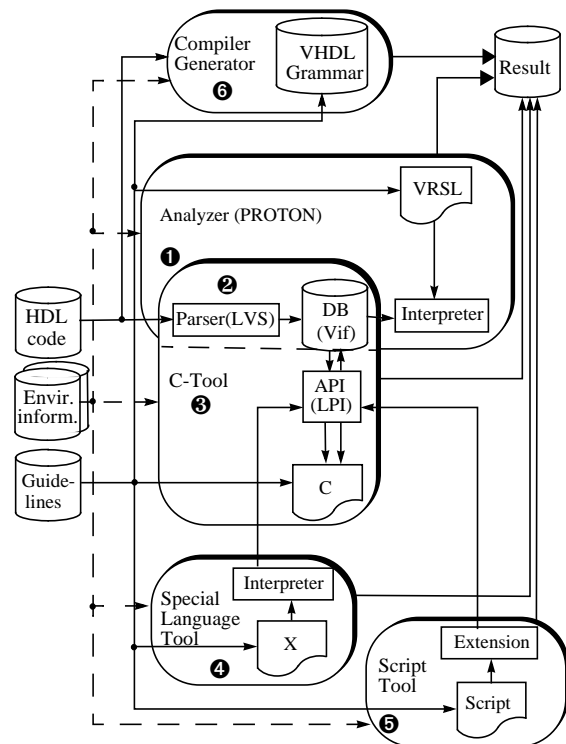


Figure 1: Analysis approaches

Fig. 1 shows the following methods:

- Use of an available commercial analysis system (e.g. PROTON of LEDA or VHDLint of InterHDL) and a special purpose languages to describe coding guidelines (e.g. VRSL for PROTON) → ①.
- Use of front-end tools, consisting of parser and database (e.g. LVS system of LEDA, AIRE of FTL) and a user interface (LPI for LVS) to the database → ②
 - a. Description of the coding guidelines with universal language (e.g. C code) → ③
 - b. Extension of ③ towards a special purpose languages utilizing an interpreter → ④
 - c. like ④, but use of a proved script language as Tcl for the specification of the coding guidelines; this approach requires the extension of the script language by access routines to the database → ⑤
- Use of a compiler generator, with which the coding guidelines are tied up directly to the grammar of the language → ⑥

For analyzer approaches of type ① and ② the possibilities to consider the additional environment information are mostly restricted. In Fig. 1 only the tools of LEDA [2] are indicated as a representative.

The described methods have different pro and cons. In the next chapter the advantages of the compiler generator method for the implementation of an analysis tool are justified.

2.2 Compiler generator approach

As shown in chapter 1.2 coding style guidelines may

- concern the grammar of a HDL,
- require context-specific information which must drawn from the environment of the HDL instance.

To analyze information of the second upstroke, there is no generally applicable approach. The kind of the related information processing strongly depends on the nature of the environment information (record, library, file system, ...). For the sake of simplicity, in the following sections environment informations are not going to be considered.

For the implementation of an analysis tool it is necessary to transform the verbal described analysis task (coding guidelines) into program code. Since guidelines concern typographical, lexical, syntactical, and semantic restrictions of the HDL grammar, compiler generator techniques offer suitable capabilities for the analyzer implementation. The following characteristics of compiler generator tools are advantageous for this purpose:

Grammar specification. The formal construction principle of the analyzed HDL is the grammar. It is e.g. specified in Language Reference Manuals (LRM in short). In compiler generator tools the grammar is available explicitly (e.g. the syntax in the scanner tool Yacc by the Extended Backus Naur Form - EBNF). Therefore, grammar rules can be used to assign suitable processing functions for the implementation of guidelines. This yields to a high transparency for the programmer.

Generation. Corresponding to the compiler generator principle the analyzer is produced automatically under usage of generator tools (e.g. standard Unix tools Lex/Yacc, C compiler).

Interpretative operation. The generated analyzer operates interpretatively and following, the compilation into an intermediate format is not necessary. Consequently, the multi-level procedure of other admissions (translation into an intermediate format → illustration of the lexis, syntax and semantic restrictions of a coding guideline in data of the intermediate format → access on to the intermediate format) is not necessary.

Profiling. Trough profiling it is possible to detect, which parts of the grammar description are passed through for a concrete HDL code segment. This provides the programmer with information about the possible „places“ for the later connection of guidelines to the grammar.

Connection. The implementation of coding style guidelines is carried out by adding processing functions at grammar rules (e.g. setting/evaluation of condition flags to trace the run through paths in the syntax tree, check of range restrictions). Compiler generator tools usually provide a interface to high level programming languages for this purpose (e.g. C code by Lex/Yacc).

Extensibility. The software of the compiler generator specification (e.g. main routine, scanner, parser, utilities) is available as source code. It is easily possible to modify the source code for further requirements. Tricky solutions can also be performed (e.g. multiple parser passes).

Tools. Because of the usage of standard Unix tools and/or GNU software the development of the analyzer does not depend of commercial tool producer.

Data capture. Similar to other approaches, the compiler generator method must also utilize temporary data structures to save informations. This data can be simple structured and be declared specific to the guidelines.

Beside the advantages there exist some disadvantages:

Maintenance. All the coding style guidelines are implemented by a common code because of there direct connection to the grammar definition. This is disadvantageous for the unambiguous maintenance of guideline (changing, deleting). By means of the interface concept explained in chapter 3.2.2 a concept to overcome this disadvantage is introduced

Information access. In alternative analysis approaches which are based on an internal database (see Fig. 1), most of the required data for evaluation are available in an already prepared form (e.g. lists). Consequently, with such approaches filter functions can be used to extract information efficiently. Nevertheless, the internal structure of databases is rather organized concerning an efficient computer-aided execution than regarding content aspects.

3 System concept of the compiler generator approach

3.1 Scanner and parser

Fig. 1 shows the main components of our software system for the coding analysis. In a prototype implementation we use the standard Unix tools Lex (scanner: lexical analysis) and Yacc (parser: syntax analysis) in a C/C++ environment. Other compiler generator tools (e.g. PCCTS, JavaCC) and other environments (e.g. Java) are suitable, too.

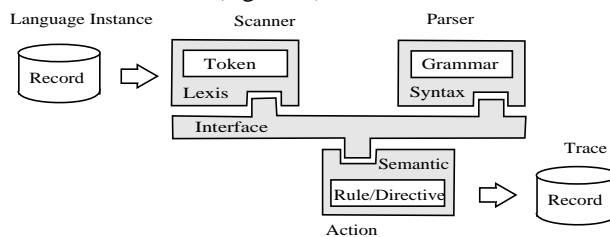


Figure 2: System conception

Within a compiler generator approach, the formal construction principle of the analyses HDL is given explicitly - namely in form of the grammar. In our prototype (Lex/Yacc) the grammar is available as EBNF. It corresponds to the HDL definition in the LRM and consists of a set of rules. The following example shows an extract from a VHDL grammar specification according to the IEEE 1076-1993 standard:

```

element_subtype_definition ::= subtype_indication
entity_aspect ::=
  entity entity_name [(architecture_identifier)]
  | configuration configuration_name
  | open

entity_class ::=
  entity | architecture
  | configuration
  | procedure | function | package
  | type | subtype | constant
  | signal | variable | component
  | label | literal | units
  | group | file

```

The specification of a grammar can be illustrated as a graph of all combinations of HDL code segments. Scanner (Lex) and parser (Yacc) organize the traversing of the relevant parts of this syntax graph for a given HDL instance. Syntax errors will be automatically identified, because there do not exist related paths in the graph. The scanner identifies the smallest unit of the HDL code (morpheme, often also called token) and qualifies them (identifier, keywords, operators, delimiter, comments, ...). Output of the parser and therefore input for the scanner is the recognized series of qualified morphemes. The scanner carries out the syntactical analysis. This is done according to the allowed sequences of morphemes, which are described by rules [3], [4], [5].

3.2 Connection of coding style guidelines

According to chapter 1.1 coding guidelines can be expressed by restrictions in use of the HDL. To implement an analyzer utilizing the compiler generator approach it is sufficient to add these restrictions at the formal language specification given by the HDL grammar. In our prototype we utilize the C/C++ interface of Lex and Yacc specifications (and the surrounding main, pre, and post routines) for this purpose. After performing such a connection of coding guidelines, the executable analyzer software is generated automatically running the Lex and Yacc tools.

3.2.1 Direct Connection

The general possibilities to connect coding guidelines to the routines of a compiler generator environment are shown in Fig. 3. The following remarks illustrate basic ideas of these assignments.

If we assume again that the syntax of a HDL grammar is interpreted as a graph, it is easy to represent HDL restrictions in this syntax graph. So, some paths or sequences in the graph could be prohibited and transitions may be

allowed only under a certain condition. Since the syntax specification is usually done by rules in a compiler generator environment, it is possible to assign the above stated steps directly to the rules inside the parser.

Similarly, the lexis part of the grammar which describes the construction of morphemes is specified by rules, too. In simple coding style guidelines, the permissible variety of a morpheme construct often is restricted only. In this case it is enough to add related checks for the restrictions to the formal definition of the morphemes inside the scanner.

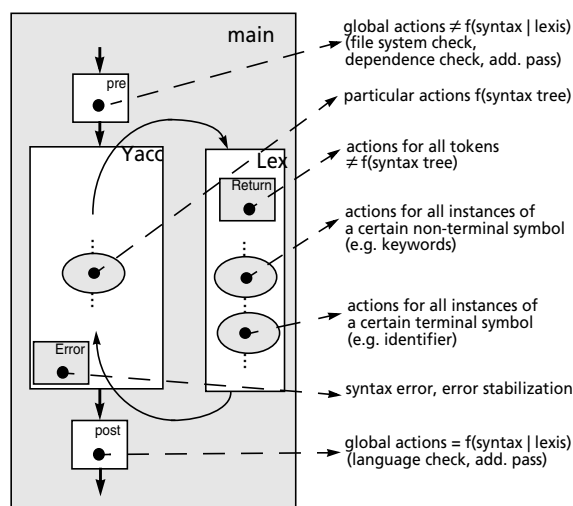


Figure 3: Connection of coding guideline lines

The following example shows the direct connection of a very simple coding style guideline to syntax rules inside the scanner. The guideline of the example says that the number of entity declarations in a VHDL instance is limited to one.

```

;entity_decl : entity_start entity_decl_1
              entity_decl_2 entity_decl_3 entity_decl_4
              t_END entity_decl_5 t_Semicolon

;entity_start: t_ENTITY
              {if(++nbrEntities >1)
               putErrorString("Error");} | bold: direct
                                                    connection

```

Benefit of the direct connection of guidelines is a short developing time. However, the direct connection is mainly suitable for a limited set of coding style guidelines. For more extensive sets the direct variant has a crucial drawback. Caused by mixing grammar rules and guideline code the analyzer software becomes almost a confused structure. Consequently, guideline maintenance (especially changing and deletion) may be more difficult. The interface concept, sketched in the next section, should help to overcome this problem.

3.2.2 Connection by a strict interface

By technological reasons, it seems advantageous to us to separate very strictly the implementation of analyzer and guideline rules [7]. In such a way, changes of the grammar

of the HDL do not immediately influence the implementation of the rules. On the other hand, changes of the rules do not always imply a re-compilation of the analyzer. Additionally, the whole system is much more structured. As a consequence, we did not have any problems to transfer the system from one platform to another. And, it is better legible than a mixed implementation of analyzer and rules.

The connection between analyzer and rules is implemented by an interface. In Fig. 4, the class-model of this concept is sketched. The module "interface.cpp" implements the corresponding interface-functions. These are fixed in an unified format for Lex-rules of the scanner as well as for Yacc-rules of the parser. These functions call the corresponding methods for filling and checking the databases. The modules "lexrules.cpp" and "yacrules.cpp" both include methods which check the guideline-rules. The class lexrules contains all those checks which are called by Lex-interface-functions and where information are transferred by Lex-rules. The analogous situation we have with respect to Yacc-rules.

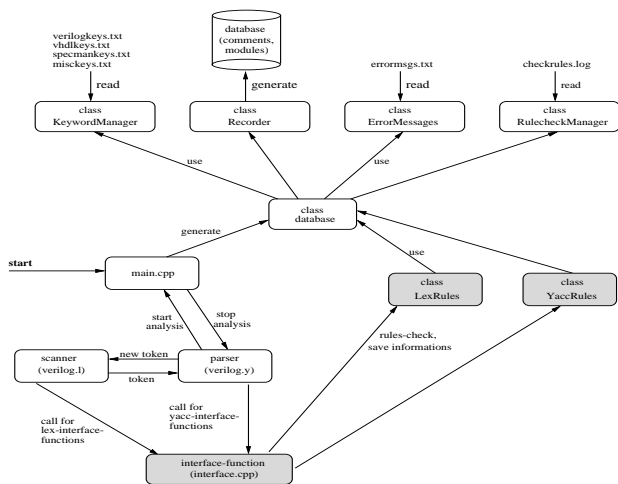


Figure 4: Interface concept

4 Implementation

Currently, we have implemented two prototypes to analyze selected coding style guidelines of industrial partners. One of them is based on a subset of the VHDL IEEE 1076-1993 grammar, the other one utilizes a grammar subset of Verilog IEEE 1364. Our analyzers produce an error file representing the detected guideline violations. A visualization software (see screen plot of Fig. 5) written in Java reads the error file and the file of the HDL instance to show the results of analysis. If coding guidelines are classified by types, the violated guideline classes can be selected easily (upper right window in Fig. 5). The link from the visualization to an editor (XEmacs) is also available in our experimental prototype.

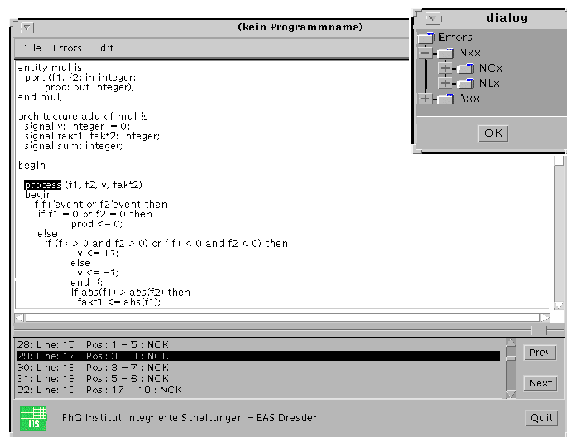


Figure 5: Visualization of analysis results

5 Conclusion

Following coding style guidelines, designers may contribute to a high quality in the process of HDL-based design reuse. We have developed an easy and flexible approach towards the analysis of HDL coding style guidelines. With the approach we exploit methods known from compiler construction. The base idea is connecting guidelines to the rules of the formal HDL grammar description. We use the C/C++ interface of proved compiler generator tools for this purpose. This universal interface also enables us to implement semantic requirements of guidelines which are not derivable from the specification means of the HDL but drawn off the environment. Prototype implementations confirm the practicability of our approach.

Future work comprises exertions to extend our approach in order to obtain analysis data required for other tasks in computer-aided reuse (like testbench generation, synthesizer control, or component retrieval).

References

- [1] Baumann, C.: „Wachstumsmarkt IP-Module“, Systeme, September 1998, pp. 66-69
- [2] LEDA homepage, URL: <http://www.leda.fr>
- [3] Bachmann, P.: „Grundlagen der Compiler-technik“, Teubner, 1975
- [4] Herold, H.: „LINUX-UNIX Profitools“, Addison Wesley, 1999
- [5] Loeper, H., Jäckel, H.J., Otter, W.: „Compiler und Interpreter für höhere Programmiersprachen“, Akademie-Verlag, 1987
- [6] Keating, M., Bricaud, P.: „Reuse Methodology Manual for System-On-A-Chip Designs“, Kluwer, 1998
- [7] Schölzel, M.: „Compiler Construction Tool“, Dokumentation of BTU Cottbus, May 1998
- [8] Actel HDL Coding Style Guide, URL: <http://www.actel.com>
- [9] XILINX Design Reuse Methodology for ASIC and FPGA Design URL: <http://www.xilinx.com/ipcenter/designreuse/index.htm>

All product names etc. mentioned in this paper are registered trademarks of their manufacturers.