A VHDL Reuse Workbench

Gunther Lehmann, Bernhard Wunder, Klaus D. Müller-Glaser
Institute for Information Processing Technology (ITIV)
University of Karlsruhe, D-76128 Karlsruhe, Germany
{leh, wun, kmg}@itiv.etec.uni-karlsruhe.de
http://www-itiv.etec.uni-karlsruhe.de/

Abstract

An increasing productivity gap affects the progress of electronic system design. As an immediately available solution, the application of reuse techniques is widely recognized. This paper presents a reuse workbench for VHDL designs which covers the three basic requirements of a design with reuse process: availability, findability, and understandability. The latter is realized by a novel reverse engineering concept for VHDL designs. It is based on the intensive use of hypertext techniques and graphical code representations.

1. Motivation

The current electronic system design process is influenced by two contradictory trends. On the one hand, the progress of semiconductor technology leads to a steadily increasing functional complexity. On the other hand, product development cycles must become smaller due to decreasing time to market periods. This contradiction can not be resolved by an enlarged design team since the portion of the so-called communication overhead rises together with the head count. Moreover, there is only a small intention within the European electronic industries to expand development budgets at the moment. Therefore, only a distinct increase of designers productivity (#transistors/man month) will remain as a reasonable solution.

The necessity of this increase is e.g. stated by [12] who predicts a growing productivity gap (Fig. 1). According to a SEMATECH study, the system complexity rose around 50% in 1995 whereas design productivity was left behind with 21% increase.

The currently proposed solutions for this productivity dilemma mainly focus on techniques which will reduce the amount of functional specification related to the implemented number of transistors. This may be achieved by different approaches:

- more abstract specifications based on the current Hardware Description Languages (HDL) followed by computer-aided transformation via high-level synthesis;
- introduction of novel (e.g. graphical) specification methods together with the development of suitable transformation tools;
- reuse of existing design data and design knowledge.

Since the first two approaches have not reached a wide commercial availability and would require huge investments in training and software, the reuse approach might remain as a cost-effective and instantly available technique to overcome the productivity problem.

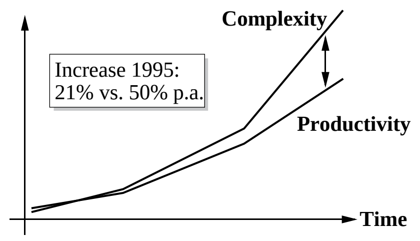


Figure 1. Productivity gap

2. From hardware to software reuse

Reuse is not new to electronic system design. Since the first commercial applications of digital electronics, standardized components and inferfaces (e.g. 74xx-ICs, TTL voltage levels) were utilized in order to reduce development and manufacturing costs. At that time, the application-specific functionality resulted from a structural configuration of standardized hardware components on a printed circuit board.

Today, electronic system design is dominated by complex and programmable components (e.g. CPLDs, FPGAs) and the usage of synthesis tools which automatically transform a technology-independent HDL specification into a low-level implementation [4].

Therefore, most of the functional properties of a design (and the design-specific intellectual property) are not kept in a structural hardware configuration but in a software-like HDL description ('software chips'). This leads to the assumption that reuse in a modern hardware design process can adopt reuse techniques originating from software engineering.

2.1. Reuse benefits

Of course, the major advantage of reuse during a software-like hardware development is the reduced design time which results from less coding effort but also from savings during the specification, documentation, and test phases. The reduced design time will directly lead to smaller development costs (up to 85%, [9]).

Additionally, the reuse of existing models will improve the predictability of design efforts, thus design risks are minimized. This benefit is also supported by an increased quality, since reused models have been 'tested' several times by different users. Last but not least, reuse might disburden engineers from repetitive tasks and therefore improve their motivation.

2.2. Reuse requirements

To fully exploit the benefits of reuse, it is mandatory that reuse becomes a design principle. The underlying basic requirements are normally separated into *design for reuse* and *design with reuse* requirements.

Design for reuse (DFR) denotes the additional effort that has to be spent during model creation in order to increase the probability of future reusage. The resulting models must show several characteristics:

- wide application range:
  - structural and functional adaptability;
  - abstraction (no application specific details);
  - portability;
- understandability:
  - deterministic behavior;
  - readability, consistent documentation;
  - encapsulation of model parameters;
- consideration of standards:
  - general and domain-specific guidelines;
  - interface structure and interface types;
- fine-grained modularity.

Design with reuse (DWR) summarizes all design activities which partly or even completely depend on previously designed models. The basic steps during a design with reuse are: selection of reuse candidates, analysis of functionality, evaluation of suitability, and adaption to the specific constraints of a design task.

The success of a DWR strategy which compensates the DFR costs, mainly depends on a wide *availability* of models intended for reuse. This is mostly achieved by a *repository* realizing computer-aided storage and management of reusable software. In addition, functional and/or non-functional retrieval mechanisms have to be provided to support the *findability* of reuse candidates. Further assistance of DWR might be contributed by computer-aided *analysis tools* which ease *understandability* and facilitate the adaption process. According to [2] the costs of modifying only 20% of an external software module are nearly the same as developing the module from scratch, if no analysis tools are applied. The analysis tools might also enlarge the confidence of a designer into external source code. This means that analysis tools could overcome the psychological conflicts ('not invented here syndrome') which usually occur during a DWR process.

3. VHDL and reuse

Severe problems appearing during the maintenance of complex electronic systems have been the major reason for developing the standardized language VHDL which allows the creation of highly reusable models. VHDL incorporates reuse techniques belonging to hardware design (structural module configuration) as well as those originating from software development (functional parametrization).

For example, the adaptability of a VHDL design is supported by generics (encapsulated inside the entity), generate-statements (conditional compilation), or unconstrained types. The portability of a VHDL design is achieved by its independence from a specific semiconductor technology and the encapsulation of platform-dependent properties. Furthermore, the standardized simulation semantics and VHDL's self-descriptiveness are supporting the analysis process. Besides these language properties which ease a DFR, VHDL also backs DWR techniques by an extensive and flexible module configuration concept.

4. A VHDL reuse workbench

The excellent reuse properties of VHDL have encouraged many users to develop and apply VHDL-based DFR strategies (e.g. [13]). This section will present a novel approach for a VHDL-based hardware design reuse workbench. The workbench should ease a DWR process in order to benefit from former DFR investments. However, the workbench might be utilized even for the reuse of any arbitrary VHDL code collection (e.g. public domain models from ftp-sites).

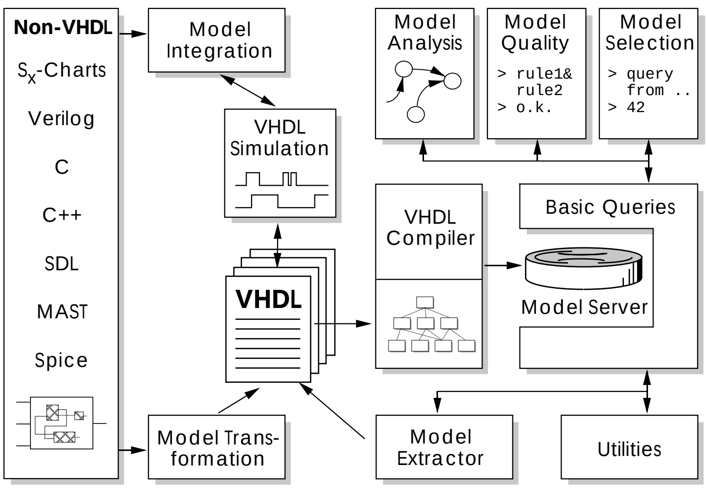


Figure 2. The VHDL reuse workbench – system overview

The workbench integrates different tools, which provide VHDL model storage, findability, functional and structural analysis as well as quality checks. In the following, we will motivate the necessity of the different workbench components as depicted in Fig. 2. The underlying concepts and the functionality of the components will be introduced.

4.1. Model server

Database: The above-mentioned requirement of a wide availability (see 2.2) is provided by a commercial object-oriented database which serves as a general *VHDL model repository*. The database comprises a client-server concept which keeps the access to the model server independent of a specific operating system or hardware platform. Locking mechanisms assure conflict-free multi-user transactions.

C++ is applied as data definition and data manipulation language. The object-orientation of C++ enables us to model complex and recursive structures, like the VHDL syntax, in an understandable and realisitic way. Furthermore, object-oriented databases allow to navigate across the data very close to its structure. This would not be possible in a relational database since they realize navigation by joins of tables.

Database modeling: To store and access VHDL designs inside the model server, a suitable representation of VHDL by a database model has to be provided. Since we will support very detailed queries (e.g. concerning signal initialization), we must store the VHDL models in a fine-grained manner. Moreover, the database model should not incorporate application details and should be extendible to future VHDL standards. Therefore, we decided to derive the *object-oriented database model* solely and systematically from the VHDL grammar.

The derivation starts from the VHDL grammar $\mathcal{G}_1$ which is defined in [7]. First, $\mathcal{G}_1$ is transformed into an abstract, object-oriented attribute grammar $\mathcal{G}_2$ [5] which facilitates the implementation of semantic attribute calculation. A succeeding, completely formal grammar translation maps grammar $\mathcal{G}_2$ to an object-oriented class hierarchy $\mathcal{G}_3$. Apart from redundant elements of chain productions, all terminal symbols and non-terminal symbols of $\mathcal{G}_1$ are represented as classes in $\mathcal{G}_3$. The inheritance hierarchy and the relationships inside $\mathcal{G}_3$ reflect the syntactic structure of $\mathcal{G}_1$. A final transformation of $\mathcal{G}_3$ simplifies the resulting database model (DBM) and adds some database-specific containers, the so-called collections. The mentioned transformations and translations take into account that all queries to the initial language $\mathbf{L}(\mathcal{G}_1)$ should be realizable, i.e.:

$$\mathbf{L}(\mathcal{G}_1) \subseteq \mathbf{L}(\mathcal{G}_2) \subseteq \mathbf{L}(\mathcal{G}_3) \subseteq \mathbf{L}(\mathbf{DBM})$$

Since the collections are attached to the VHDL-specific classes, syntax based queries can be applied across all VHDL models inside the repository, e.g. "check in all design units if signals are explicitly initialized" (*bottom-up query*). Besides this, each design unit might be queried from the class library_unit down to its terminal classes (*top-down query*).

VHDL compiler: Before VHDL source code files are transfered to the model server, the code structure will be automatically analysed, i.e. the design hierarchy is extracted by a modified makefile generator, which was initially provided by [15]. After this *structural decomposition*, a *VHDL compiler* generates instances of classes and inserts relationships and derivations in order to reflect the VHDL source code. The VHDL compiler is based on a LALR(1) grammar and is implemented with the Cocktail compiler generator [5].

4.2. Model analysis

A core functionality of the reuse workbench is provided by the model analysis module implementing novel VHDL reverse engineering techniques. These techniques are aimed to simplify the *functional* and *structural analysis* of VHDL designs. It is expected that the model analysis module will accelerate the process of adaption and will overcome the mentioned psychological DWR conflicts (see 2.2).

The different, mostly graphical user interfaces (Fig. 3) will be dynamically generated on request. They could invoke each other, as it is indicated by the straight lines in Fig. 3. Besides this, many automatically attached hyperlinks provide cross references between the details of a VHDL design (curved lines). Some screenshots of the implemented analysis interfaces are published in [10].

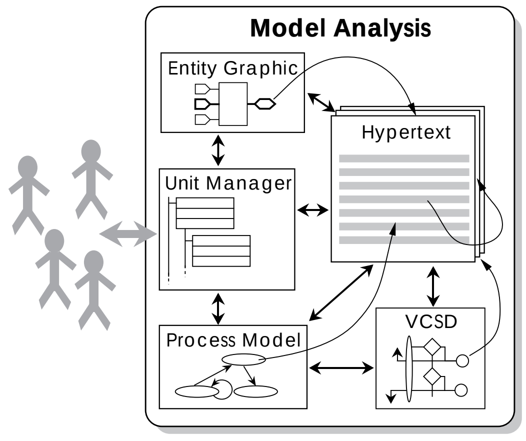


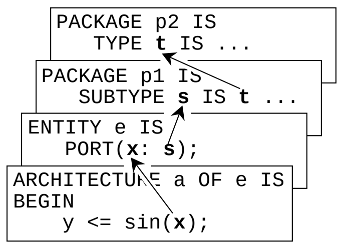
Figure 3. Model analysis interfaces

The unit manager serves as an entry point to the model analysis. It visualizes the complete hierarchy of a VHDL design and offers some navigational aid because it highlights the current position during the analysis and tags already analysed design units. The entity graphic depicts the interface of each VHDL entity (ports and generics).

The remaining analysis interfaces (Hypertext, Process Model Graph (PMG), and VCSD) are aimed to attack the understanding problems which typically occur during VHDL source code analysis. They will be discussed in detail in the following paragraphs.

VHDL hypertext: The outstanding structuring capabilities of VHDL allow a modular, redundancy-free and reusable description of even very complex digital systems. Unfortunately, structuring and partitioning accross many design units and source code files will prevent from quickly analysing a single model, since related information is widely distributed (Fig. 4).

Therefore, the model analysis module provides a hy-



```
PACKAGE p2 IS
   TYPE t IS ...
PACKAGE p1 IS
   SUBTYPE s IS t ...
ENTITY e IS
   PORT(x: s);
ARCHITECTURE a OF e IS
BEGIN
   y <= sin(x);
```

Figure 4. Example of distributed information

pertext interface, which enables the user to simply explore a VHDL design by mouse clicks ('VHDL surfing'). For that, hyperlinks are automatically attached to all identifiers inside the VHDL source code. The hyperlinks point to the correlating declaration of each identifier (Fig. 5). Since every identifier can be used many times, a directed many-to-one relation exists for each identifier declaration. The declaration itself and the corresponding design unit also contain many hyperlinks which point to further declarations, and so on.

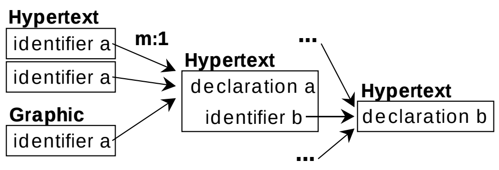


Figure 5. VHDL hypertext structure

Because the hyperlinks are attached *dynamically* on user's request, we are able to implement a hyperlink web which is *interactively configured* by the user. This means that the user might select links by VHDL-specific criteria (e.g. links to signal declarations or links to input port declarations). A solely activation of input port links e.g. will enable the user to quickly detect if and where input ports are read inside an algorithmic VHDL description. After selection, a simple mouse click will immediately open the design unit where the identifier is declared; the corresponding declarative statement will be highlighted.

```
CONFIGURATION c OF e IS
 FOR adder
  FOR c1: ha USE
   ENTITY work.ha(beh);
   FOR beh
     FOR x1: xor2 USE
     ENTITY work.x(beh);
  ...
END c;
```

Figure 6. Hyperlinks inside a VHDL configuration

Besides this, the user might navigate along an already selected link path through different design units via a back/forward mechanism. For example, it is possible to access the declarations of all bold-faced identifiers inside configuration c (Fig. 6) by a mouse click and return to configuration c each time.

VHDL process model graph: The behavior of a VHDL model is determined by its concurrent statements inside the VHDL architecture. A manually carried out functional analysis has to identify the signal drivers and triggering conditions (sensitivity sets) of all concurrent statements. Often, the sensitivity set must be extracted from complex expressions inside the wait-statements which might be hidden inside some procedures. A further analysis problem results from the fact that concurrency is described by a linear, one-dimensional medium (ASCII text files). Therefore, the designer has to move up and down inside the source code during the analysis process (Fig. 7). This procedure is tedious and error-prone (cf. goto programming).

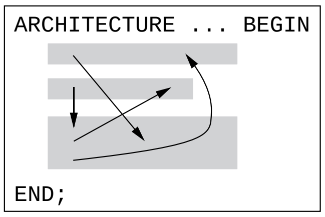


Figure 7. Non-sequential functional analysis

The VHDL reuse workbench supports the functional analysis by an automatic translation of VHDL architectures into a meaningful process model graph[1] (PMG). The nodes of this graph represent the concurrent statements; the edges depict the activating signals. Statements which might invoke themselves via one or more signals (or inout ports) are marked by a loop. Furthermore, redundant signals are eliminated

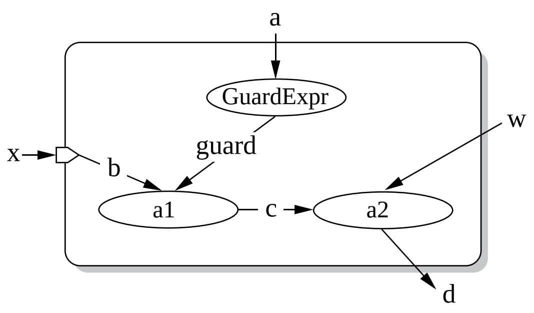


Figure 8. Process model graph (PMG) of block b1

from the PMG. This happens e.g. if signals inside a sensitivity set have no corresponding signal driver. Therefore, the PMG might also be applied during the debugging phase of a VHDL design project. Connections

[1]A more dataflow oriented diagram with identical naming was introduced by [1].

to the VHDL hypertext interface are provided by automatically attached hyperlinks which point from each identifier inside the PMG (port, signal, label) to the correlated declaration or usage inside the VHDL source code.

The hierarchical 'clustering' of concurrent statements inside a generate- or block-statement is visualized by hierarchical diagrams. Fig. 8 depicts a PMG which has been extracted from the following piece of VHDL code. Please note that the signal guard which is implicitly declared by the guard expression was made visible to the user.

```
       ...
b1: BLOCK (a = '1')
  PORT (b: IN bit);
  PORT MAP (b => x);
  SIGNAL c: bit;
BEGIN
  a1: c <= GUARDED b;
  a2: nand2 PORT MAP(c, d, w);
END BLOCK b1;
       ...
```

VHDL control structure diagram: Besides the PMG (visualizing concurrent statements), the model analysis provides automatically generated diagrams which depict the structure of *sequential code* inside processes and subprograms (functions and procedures). The diagram is based on so-called control structure diagrams (CSD) which were introduced by [3] to ease ADA source code analysis. Compared to the well-known structure charts and flow diagrams, CSDs show a strong relation to the source code, allow an easy implementation, and efficiently visualize exit- and next-statements. Since CSDs only expand in two directions (to the right: structural depth, to the bottom: code sequence) we have been able to easily extend their functionality by a user-configurable depth of visualization.

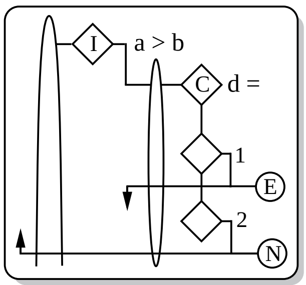


Figure 9. VHDL control structure diagram (VCSD)

Moreover, we added the following properties, resulting in the VHDL control structure diagram (VCSD):

- visualization of VHDL specific constructs;
- user-defined amount of depicted information;
- automatically inserted hyperlinks pointing from all structures and identifiers to the VHDL hypertext source code.

Fig. 9 shows a small example of a VCSD containing two loops, one if-, one case-, one exit-, and one next-statement.

4.3. Model quality

The expressive power and flexibility of VHDL leads to a large number of possible modeling styles. This fact has motivated the development of tools which support an automatic inspection of modeling quality, e.g. [14].

The VHDL reuse workbench incorporates a model quality checker which processes rules concerning synthesizability and simulation efficiency as well as reusability in order to support the DFR process. Different reuse facets like understandability, adaptability, portability, and deterministic behavior are inspected as far as possible.

Mostly, rules depend on the specific application (e.g. PCB vs. ASIC), the designer's company, or the applied design tools. For this reason, the predefined rules can be parametrized and configured to rule sets inside the individual environment.

4.4. Non-VHDL modeling domains

Besides the digital world, analog components and the system environment have to be regarded if system modeling and simulation are performed. Therefore, we decided to provide some interfaces which could reuse non-VHDL models via *transformation* (i.e. converting them to VHDL) or at least by coupling them to a VHDL simulator via model *integration* (Fig. 2). Currently, we complement Verilog-to-VHDL translators and VHDL backends of system design tools by a VHDL interface which transforms control system models of xmath [8] into VHDL models. An additional flexible interface allows to connect external simulation models, written in the programming language C, to a VHDL simulator via UNIX interprocess communication [11]. In the future, this interface might be replaced by the Open Model Interface (OMI) which is currently designed by the Open Modeling Forum (OMF) [6].

5. Conclusions

This paper presents a reuse workbench for VHDL designs simplifying code reuse in order to narrow the existing productivity gap. First applications proved that especially the wide availability of VHDL models and the automatic analysis tools will motivate designers to regard foreign VHDL models. Additionally, the workbench can reduce communication overhead inside larger design teams and may support code review and error-detection.

Since the tool targets VHDL code and VHDL related design data, it will not cover all reuse aspects of electronic design (e.g. layout macros). At the moment, a second limitation results from the simple model selection via a non-functional key word search. Furthermore, the tool is based on a compiler independent of compilers interfacing VHDL simulators. This leads to extra compilation times during the design process.

Future developments will include an interactive rule correction functionality inside the quality checking module. Additionally, the hypertext interface should become editable (with an update feature) to ease the reuse phase of model adaption. Finally, the model selection module might be enriched by a formalized, VHDL-oriented query language.

References

[1] J. R. Armstrong. *Chip-level Modeling with VHDL.* Prentice-Hall, Englewood Cliffs, NJ, USA, 1989.

[2] B. W. Boehm. Keynote speech. In *ACM Computer Science Conference*, Phoenix, AZ, USA, 1994.

[3] J. H. Cross. Reverse Engineering Control Structure Diagrams. In *IEEE Working Conference on Reverse Engineering*, Baltimore, MD, USA, 1993.

[4] E. Girczyc and S. Carlson. Increasing Design Quality and Engineering Productivity through Design Reuse. In *30th ACM/IEEE Design Automation Conference*, Dallas, TX, USA, 1993.

[5] J. Grosch. *Cocktail – Toolbox for Compiler Construction.* Karlsruhe, Germany, 1994.

[6] W. Hobbs. Model Availability, Portability and Accuracy. In *Workshop on Libraries, Component Modeling, and Quality Assurance*, Nantes, France, 1995.

[7] *IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1987.* New York, NY, USA, 1988.

[8] Integrated Systems Inc., Santa Clara, CA, USA. *MATRIX$_x$ Product Family Reference Manuals*, 4.2 edition, 1995.

[9] R. G. Lanergan and C. A. Grasso. Software Engineering with Reusable Designs and Code. In *IEEE Tutorial: Software Reusability.* IEEE Computer Society Press, Washington, D.C., USA, 1987.

[10] G. Lehmann, B. Wunder, and K. D. Müller-Glaser. VYPER! – A VHDL Hypertext Environment for System Design Reuse. In *VHDL International Users' Forum*, Newton, MA, USA, 1995.

[11] G. Lehmann, B. Wunder, M. Wolff, and K. D. Müller-Glaser. An Environment for Electronic Systems Simulation based on VHDL. Nantes, France, 1995.

[12] F. Musa. VHDL and Verilog: Who Needs Them? In *VHDL International Users' Forum*, Newton, MA, USA, 1995.

[13] V. Preis, R. Henftling, S. März-Rössel, and M. Schütz. A Reuse Scenario for the VHDL-Based Hardware Design Flow. In *EURO-VHDL*, Brighton, UK, 1995.

[14] H. Sahm, C. Mayer, J. Pleickhardt, J. Schuck, and S. Späth. VHDL Development System and Coding Standard. In *33rd ACM/IEEE Design Automation Conference*, Las Vegas, NV, USA, 1996.

[15] H. M. Thaker. *Automatic Generation of Makefiles for VHDL – Application Note.* Bell-Northern Research VHDL Group, USA, 1991.