Basic Concepts for an HDL Reverse Engineering Tool-Set

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

Designer's productivity has become the key-factor of the development of electronic systems. An increasing application of design data reuse is widely recognized as a promising technique to master future design complexities. Since the intellectual property of a design is more and more kept in software-like hardware description languages (HDL), successful reuse depends on the availability of suitable HDL reverse engineering tools. This paper introduces new concepts for an integrated HDL reverse engineering tool-set and presents an implemented evaluation prototype for VHDL designs. Starting from an arbitrary collection of HDL source code files, several graphical and textual views on the design description are automatically generated. The tool-set provides novel hypertext techniques, expressive graphical code representations, a user-defined level of abstraction, and interactive configuration mechanisms in order to facilitate the analysis, adoption and upgrade of existing HDL designs.

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 solved by an enlarged design team because the portion of the communication overhead will rise together with the head count. Therefore, only a distinct increase of designer's productivity (#transistors/man month) will remain as a reasonable solution. The necessity of this increase is e.g. stated by [14] who predicts an enlarging productivity gap. According to a SEMATECH study, the system complexity rose 58% in 1995 whereas design productivity was left behind with 21% increase [9].

Besides the reduction of tool execution times, the currently proposed solutions of this productivity dilemma focus on techniques which will reduce the amount of functional specification that has to be generated by the design team. This may be achieved by more abstract specifications based on the current HDLs, by novel graphical specification methods, or by the reuse of existing design data and design knowledge. Since the first two approaches mainly depend on efficient transformation tools which have not reached a wide commercial availability, the reuse approach will remain as an instantly available and cost-effective technique to overcome the productivity dilemma.

2. From hardware to software reuse

Reuse is not new to electronic system design. Since the early commercial applications of digital electronics, standardized components and interfaces (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 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 [6]. 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").

In general, the major advantage of software-based reuse 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. Former empirical studies in the area of software development have shown that consequent reuse may decrease design expenses from 32% up to 85% [8].

ICCAD '96
1063-6757/96 $5.00 © 1996 IEEE

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, because reused models have been "tested" several times by different users. Last but not least, reuse can disburden engineers from repetitive tasks thus improving their motivation.

In order to benefit from these reuse advantages, it is necessary that a design with reuse process is assisted by computer-aided *reverse engineering tools* which ease understandability of existing designs and facilitate an adaption process. According to [3] the costs of modifying 20% of an external software module are nearly the same as developing the module from scratch, if no reverse engineering tools are applied. These tools may also enlarge the confidence of a designer into the qualification of external source code modules. This means that analysis tools can ease the psychological conflicts ("not invented here syndrome") which usually occur during a reuse process. The following section will demonstrate why reverse engineering tools are of particular importance in the case of VHDL "software".

3. VHDL and reverse engineering

Hardware description languages usually incorporate syntactical constructs which allow the development of widely applicable modules (e.g. defparam in Verilog). Especially during the conception of VHDL, reusability was recognized as a major topic because VHDL should overcome typical upgrade problems during the maintenance of complex electronic systems.

VHDL adopts reuse techniques which were derived from PCB design techniques. Language constructs like port map, configuration, library, or open support a flexible structural module selection and wiring [7]. Additionally, VHDL backs reuse concepts originating from software development (functional parametrization). For example, the functional adaptability of a VHDL design is supported by generics (module parameters), generate (conditional compilation), or unconstrained types. In [16] it is shown that even complex components like DRAM controllers, microprocessor interfaces or CRC[1] generators can be implemented as universal VHDL modules.

Unfortunately, the capabilities of VHDL lead to the following analysis problems which will aggravate the reverse engineering of existing VHDL designs:

Syntactical Complexity: In order to support algorithmic modeling, VHDL offers mostly all language constructs incorporated in common sequential programming languages, like C or PASCAL. Among these constructs are branches (if, case), for and while loops with exit and next statements, as well as structuring capabilities (function, procedure). Additionally, the language is strongly enriched by constructs which serve for hardware-specific modeling, e.g. model semantics depending on simulation time (signal, after), physical units, logical operators, or hardware-oriented port modes (buffer). Therefore, the VHDL grammar includes much more production rules and terminal symbols than the grammar of C or PASCAL. A complementary language extension is caused by some redundant modeling capabilities. For example, concurrent signal assignments can be replaced by a functional equivalent process statement. In summary, the extensive language capabilities allow very different coding styles thus preventing a quick analysis of VHDL source code.

Structuring Capabilities: Besides the common structuring capabilities via functions and procedures, VHDL offers additional structuring by five design units and concurrent statement clustering via block and generate statements. Considering also the process statement, VHDL'93 has ten different declarative regions. These outstanding structuring capabilities allow a modular, redundancy-free and reusable description of even very complex digital systems. Unfortunately, structuring and partitioning across many design units will prevent from quickly analysing a single model because related information is widely distributed. A small example taken from the ALU of the DP32 processor [2] illustrates this restriction of reverse engineering (Fig. 1). Additionally, the designer has to

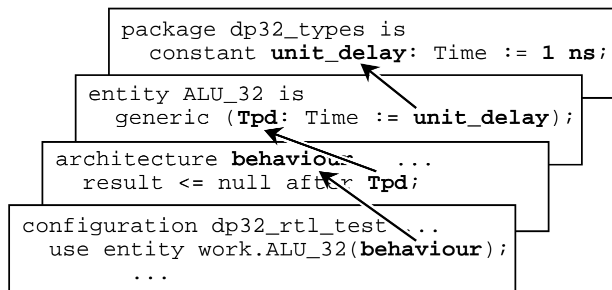

Figure 1. Distributed information

keep in mind the VHDL rules for scope and visibility during the analysis process. Further difficulties result from the fact that there is a one to many relation between a VHDL source code file and the included VHDL design units, and that the unit identifiers are totally independent from file names. Finally, the designer has to regard the context of an overloaded subprogram call, if he wants to identify its implementation.

Concurrent Statements: The behavior of a VHDL model is determined by its concurrent statements inside the VHDL architecture. During a manually carried out reverse engineering analysis, the designer will primarily have to identify the signal drivers

[1] Cyclic Redundancy Code

and triggering conditions (sensitivity sets) of all concurrent statements. Often, the sensitivity set must be extracted from complex expressions inside multiple wait statements which additionally might be hidden inside some procedures. A further analysis problem results from the fact that the 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. 2). This procedure is tedious and error-prone (cf. goto programming style).

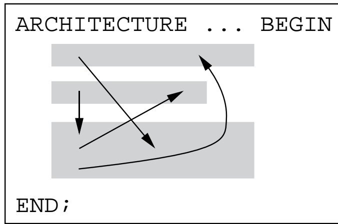

Figure 2. Non-sequential functional analysis

Simulation Semantics: The mentioned problems during a functional analysis are aggravated by VHDL's complex execution semantics, which can not be derived from common programming languages. This includes e.g. the delta cycle or the preemption mechanism.

Future enrichments of the VHDL standard (e.g. VHDL-AMS) will enlarge the reuse problems which are imposed by VHDL's complexity. Thus, computer-aided HDL reverse engineering tools will play a more and more important role during the design process of electronic systems. However, the application of these tools need not to be restricted to pure reuse. They are useful whenever complex HDL code has to be analysed and modified, e.g. system maintenance, code review, error detection, large project management, design documentation.

4. A VHDL reverse engineering tool-set

4.1. Basic requirements

A VHDL reverse engineering tool-set has to support a designer during the functional and structrual analysis of VHDL source code in order to ease the reuse of design knowledge. Several conceptual requirements must be met by such a tool-set:

Generality: To achieve a large reuse potential, the tool-set should be able to handle any syntactical correct VHDL code collection. An adoption towards a specific modeling style, a VHDL subset, some special formatting rules, or a distinct modeling domain (e.g. finite state machines) must be avoided.

Automatic Generation: The tool-set should encourage a designer to benefit from former design activities. Therefore, the tool-set must provide the extracted information fast and without manual interaction. This means for example that the design hierarchy and compilation dependencies must be derived automatically from any arbitrary VHDL source code collection.

Complexity Management: In spite of VHDL's complexity, the tool-set has to represent even large VHDL designs in a compact and plausible manner. This can be accomplished by hierarchically organized code representations. Additionally, the tool-set has to offer flexible configuration mechanisms which allow to define the amount of visualized code, the displayed scope, and the degree of detail. Furthermore, the representations should avoid redundancies. Finally, a superior tool manager has to provide some navigational aid.

Intuitive Representations: The derived VHDL code representations must be comprehensible and nonambiguous. This requirement can be met by graphical symbols, which are similiar to those of well-known graphical languages (e.g. flow charts). Furthermore, the graphical symbols should provide references to the corresponding source code sections. An intuitive textual code representation may be obtained, if the interface adopts usage concepts from tools which are familiar to the designer (e.g. WWW browsers). On the other hand, all notions inside the tool-set interfaces should correlate to VHDL-specific terms.

Interactive Visualization: Interactivity will motivate a designer to step into external VHDL source code. Therefore, graphical and textual code representations have to include dynamic and user-controlled displaying capabilities.

4.2. General structure

In the following, we present the concept of the VHDL reverse engineering tool-set VYPER! (VYPER! = Vhdl hYPERmedia) which fulfills the above-mentioned general requirements. The basic structure of VYPER! is depicted in Fig. 4. A reverse engineering process starts with a *structural analysis* of the VHDL source code. After this, the VHDL design units are compiled in the hierarchical correct order. Then the user may select from five different interface types to analyse the VHDL code. The VHDL modeling domains corresponding to the different interface types are shown in Fig. 3. On the top of the VHDL-specific hierarchy of abstraction, the unit manager shows the composition of an entire design by design units. Inside the architecture units, a process model graph (PMG) visualizes the control flow of the concurrent statements. Closest to the source code is the VHDL Control Structure Diagramm (VCSD) which depicts the control structure of sequential descriptions. A more general role is covered by the VHDL hypertext interface because it is applica-

ble for all VHDL design units. In contrast to common hypertext interfaces inside some CASE software tools, we provide an interactive and language-specific configuration of the underlying hyperlink web.

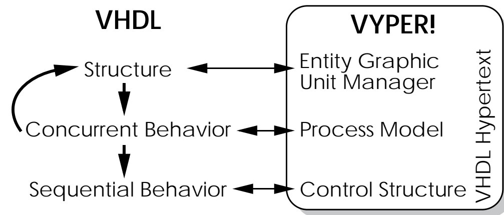

Figure 3. VHDL application domains of the VYPER! interfaces

Most of the interfaces can invoke each other, as it is indicated by the straight arrows in Fig. 4. Furthermore, there are many, automatically inserted *hyperlinks* which bind together interrelated information and avoid redundancies (curved arrows in Fig. 4).

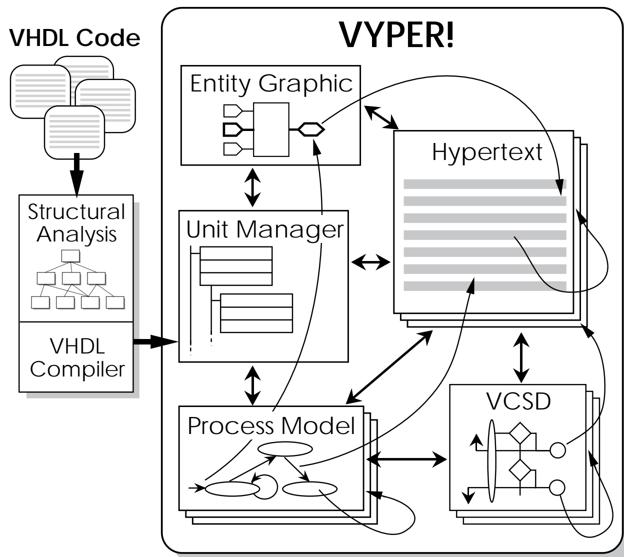

Figure 4. VHDL reverse engineering tool-set VYPER!

The unit manager serves as entry point to model analysis. It visualizes the complete hierarchy of a VHDL design and gives some *navigational aid*, because it highlights the current position during the analysis process and marks up already analysed design units. The entity graphic depicts the interface of each VHDL entity (ports and generics). The other analysis interfaces (VHDL Control Structure Diagram, Process Model Graph, and VHDL Hypertext) offer *hierarchically organized code representations* and are widely *user-configurable*. They are aimed to attack the mentioned analysis problems which typically occur during a VHDL reverse engineering process. These interfaces will be discussed in detail in the following sections.

5. VHDL control structure diagram

The algorithmic descriptions of a VHDL design are kept in processes and subprograms (functions and procedures). Inside these statements, the designer might apply all usual syntactical constructs of common sequential programming languages. In general, the analysis of such algorithmic descriptions can be supported by automatically generated diagrams which primarily visualize the algorithmic structure. The following three basic visualization techniques are well-known in the area of software-engineering:

Box Technique: The different language constructs are depicted by rectangular polygon symbols. Nested polygons visualize a syntactical hierarchy. A sequence of statements is shown by vertically aligned polygons, whereas horizontal alignment illustrates alternatives. The specific properties of each polygon are indicated by fragments of source code inside the polygons. Most of these box technique diagrams (e.g. Lindsey charts [11], [17]) are derived from the familiar Nassi-Shneiderman diagrams [15], [13].

Box and Line Technique: This technique is mostly known as flowchart diagram. It utilizes a few expressive graphical symbols representing statements, loops, branches, and subprogram calls. Their sequence of execution is depicted by directed edges between the symbols [4], [17].

Line Technique: The basic idea of this technique is to emphasize the syntactical structure of a previously formatted source code by simple brackets on the left side of the code ("action diagrams", [12]). An ADA-specific variant of this technique was introduced by [5] (Fig. 5).

The first and second technique are typically applied during the initial software design. Unfortunately, they are not suited for an automatic generation inside a reverse engineering tool, because they will require time-consuming drawing algorithms in order to generate compact diagrams for complex VHDL designs. Additionally, these techniques show a weak relation to the underlying source code because they transform the linear (one-dimensional) code into a two-dimensional representation. Furthermore, the graphical symbols do not provide enough space for source code quotations.

Therefore, VYPER! visualizes sequential VHDL code by a technique which is derived from the one-dimensional CSDs. The resulting VCSDs (VHDL Control Structure Diagrams) are enriched by the following basic properties in order to meet the general requirements mentioned in section 4.1:

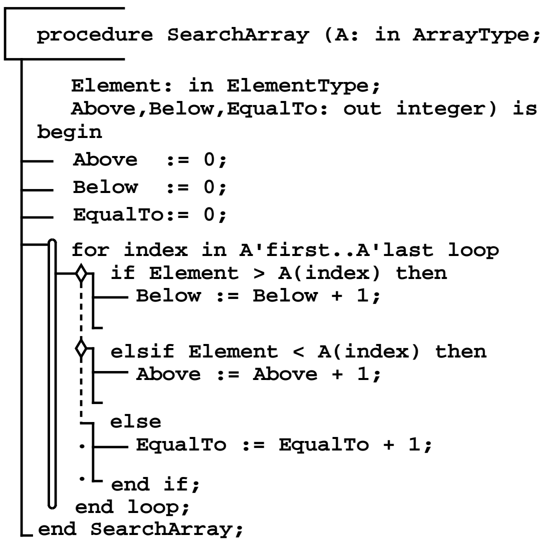

Figure 5. Example of an ADA control structure diagram

- All language constructs are depicted by *intuitive* and *VHDL-specific* graphical symbols (Fig. 6, Fig. 7).

- VYPER! should avoid redundant representations and should be able to handle any VHDL code formatting. Therefore, the source code is not completely inserted in a VCSD. Automatically integrated *hyperlinks* are used instead. A hyperlink selection will exactly *highlight* the corresponding source code fragment (Fig. 8) inside the VYPER! VHDL hypertext interface (section 7). Only the relevant expressions and identifiers are automatically extracted from the source code to complement the graphical symbols inside the VCSD (Fig. 9).

- The VCSD can be *interactively configured* by the user to support the analysis of complex designs. Sequential statements like if, case, or loop statements, which might contain further statements can be expanded or collapsed interactively and independently (Fig. 10). Beyond this, the user can select between a single or multi window mode and can interactively hide or activate all textual information inside the VCSD. Furthermore, the number of hierarchical levels which are displayed inside one VCSD is adjustable. A larger number will cause the creation of a new diagram ("detail view"). Finally, the VCSD of a procedure can be simply invoked by a hyperlink inside the procedure call symbol.

- Zoom and scroll sliders as well as a back/forward functionality also support the handling of large designs. The latter allows to walk through the history of already analysed diagrams.

- In order to ease the analysis of algorithmic descriptions based on nested loops, the destination of a next or exit statement is indicated by a horizontal arrow (Fig. 9, right side of Fig. 10). The existance of one or more exit and next statements is visualized, even if they are hidden inside collapsed symbols (left side of Fig. 10).

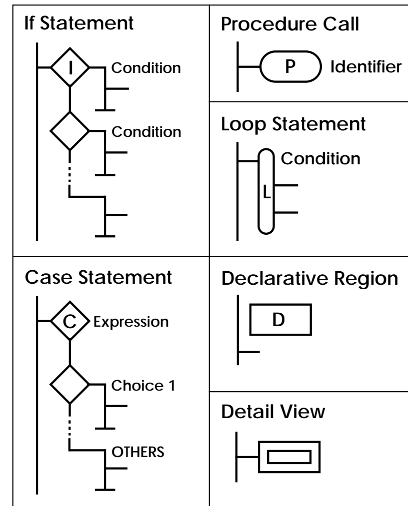

Figure 6. Non-terminal VHDL symbols inside VCSD

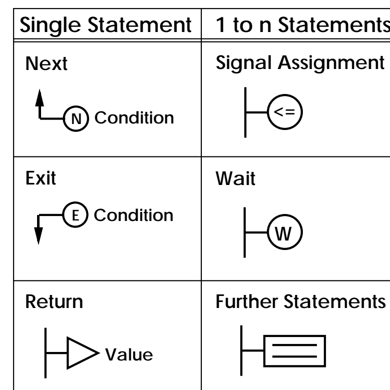

Figure 7. Terminal VHDL symbols inside VCSD

Fig. 9 shows the implementation of the VCSD interface. Via two short cut buttons named "Open Flaps" and "Close Flaps", a diagram can be fully expanded or collapsed instantly. The depicted fully expanded example was generated from the badly formatted code fragment shown in Fig. 8. The VCSD quickly clarifies that the last signal assignment is located inside the outer loop. Additionally, the corresponding

```
FOR i IN 1 TO 5 LOOP

    IF a = 10 THEN EXIT; end if;
        FOR k IN 0 TO 3 LOOP
            IF b < 10 THEN
NEXT;
    ELSE
        check(b);
        WAIT ON start;
    END IF;

END LOOP;
count <= b;
```

Figure 8. Automatic highlight of statements inside the VYPER! VHDL hypertext

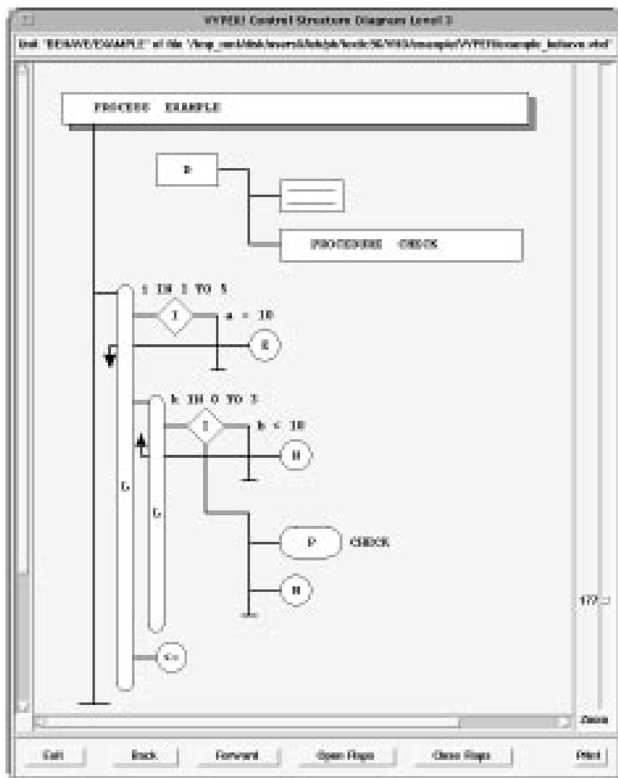

Figure 9. VHDL Control Structure Diagram VCSD

loops of the exit and next statements are easily identified within the VCSD. The highlight function allows to exactly detect the source code fragment of a language construct. Fig. 8 shows the highlighted fragment inside the VHDL hypertext module, if the second `I` rhomb is selected.

6. VHDL process model graph

Besides the VCSD module, the reverse engineering tool-set comprises an interface which illustrates the *dynamic behavior* of the concurrent statements. This is realized by an automatic translation of each VHDL architecture into a comprehensible process model graph[2] (PMG). The nodes of the graph represent the concurrent statements. Different kinds of statements (e.g. signal assignment, process, or procedure call) are distinguishable by different outline colours. Fig. 11 depicts the PMG which has been extracted from the VHDL architecture shown in Fig. 13.

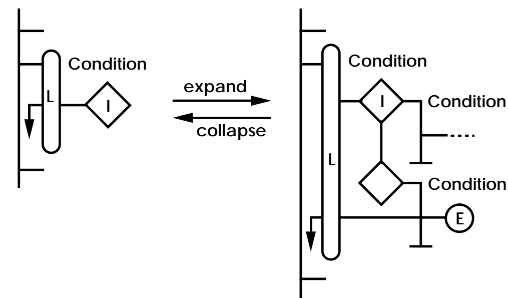

Figure 10. Visualization of a hidden EXIT statement

The edges attached to the nodes depict signals which may activate concurrent statements and which are driven by other statements. Statements which might invoke themselves via one or more signals (or inout ports) are marked by a loop (e.g. port signal `data` in Fig. 11). A loop without any signal identifier represents a concurrent statement that has no trigger conditions (e.g. node `xor_2` in Fig. 11). Nodes like this one simply indicate that no other concurrent statement will have a chance to execute. In contrast to this, a node without any arriving edge depicts a concurrent statement which is invoked only once during the initialization phase. Furthermore, redundant signals are eliminated from the PMG. This happens e.g. if signals inside a sensitivity set have no signal driver (e.g. signal `clk`, see VHDL source code example in Fig. 13). Therefore, the PMG is also valuable during the *debugging* phase of a VHDL design process.

A connection to the VHDL hypertext module is provided by automatically attached *hyperlinks* which point from each identifier inside the PMG (port, signal, label) to the corresponding declaration or statement inside the VHDL source code. If the node represents a process or a concurrent procedure call statement, it is also possible to directly activate the VCSD of this statement.

The hierarchical "clustering" of concurrent statements inside a generate or block statement is visualized by dynamically generated, *hierarchical diagrams* in single or multi window mode. Renaming of signals via port maps or bus splitting via some variables of the generation_scheme [7] are taken into account. Please note in Fig. 11 that the signal `guard` which is implic-

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

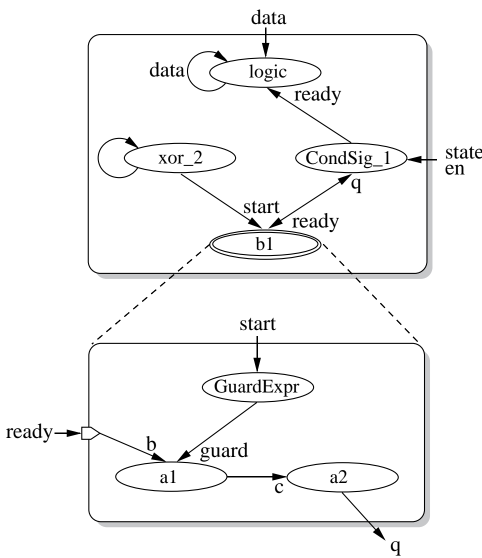

Figure 11. Hierarchical process model graph PMG

itly declared by the guard expression (block statement in Fig. 13) has been made visible to the user. In order to handle complex designs, the user can again configure the displaying properties interactively. For example, it is possible to solely visualize the control flow of the non-structural parts of a VHDL architecture (i.e. component instantiation nodes are hidden).

7. VHDL hypertext interface

In order to overcome the mentioned problems caused by VHDL's structuring capabilities, VYPER! includes a module realizing a novel VHDL hypertext concept. It enables the user to simply explore distributed information inside 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 declaration of each identifier (Fig. 12). Since every identifier can be used many times, a directed many to one relation is introduced for each identifier declaration. The referenced declaration and the corresponding design unit may of course contain again many hyperlinks, which point to further declarations, and so on.

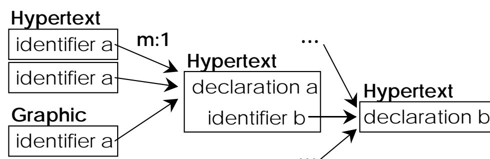

Figure 12. VHDL hypertext concept

Since the hyperlinks are only 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 is not confused by an unwieldy amount of hyperlinks because he can activate particular link types by VHDL-specific criteria (e.g. links to signal declarations, links to port declarations). A solely markup of port links e.g. will enable the user to quickly detect if and where ports are used inside an algorithmic VHDL description. After the link activation, a simple mouse click will immediately open the design unit where the identifier is declared. The corresponding declarative statement will be highlighted (like `clk` in Fig. 13).

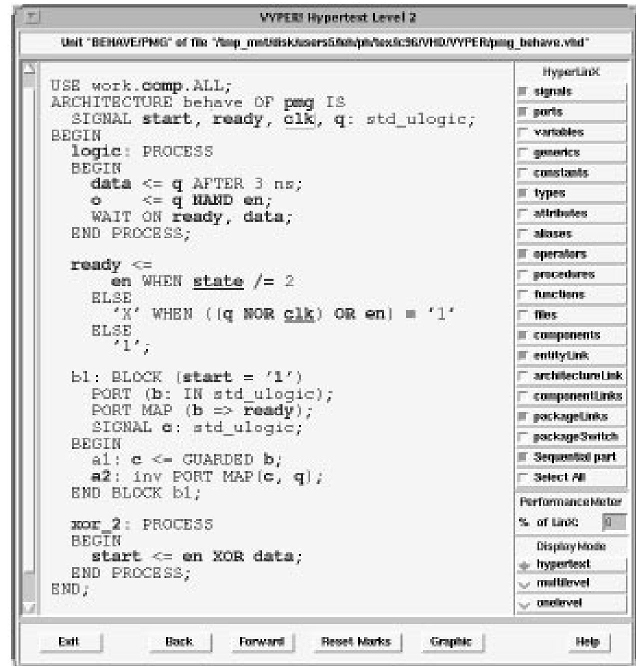

Figure 13. VHDL hypertext interface

The implementation of the VHDL hypertext interface is shown in Fig. 13. To facilitate the usage, we partly adopt the design and functionality of WWW browsers. For example, the user can navigate along an already selected link path through different VHDL design units by back and forward functions. The hyperlinks are directly embedded into the source code (boldfaced identifiers in Fig. 13) and are marked, if they have been selected (here: underlining). The VHDL-specific selection of hyperlink classes is realized via the select bar on the right side. An additional class of links named "Sequential part" is aimed to provide an easy hyperlink access to the VCSD of processes and subprograms.

8. Experimental results

The presented reverse engineering concepts have been realized by a fully functional evaluation protoype

based on TCL/TK and a commercial VHDL compiler (an optimized object-oriented VHDL compiler is under development [10]). The execution times on a SunSparc10 workstation are in an acceptable range (see processor benchmarks in Table 1). Merely the small generation performance of approximately 100 links/sec. leads to larger response periods inside the hypertext interface, if lengthy VHDL models are analysed.

Processor model	**DLX**	**DP32**	**R6502**
Lines of code	9630	1413	4921
No. of design units	72	31	59
Initial compilation of entire design	9,9	2,5	7,4
Creating unit manager	2	< 1	1
Activation of hyperlinks inside ALU	4	4	2
Creating other graphical interfaces for ALU	< 1	< 1	< 1

Table 1. Performance data of VYPER! in CPU sec.

In order to evaluate the analysis capabilities of VYPER!, we have passed a set of questions on unkown VHDL designs to different users (e.g. "Identify the sequence of controller states during the execution of command xyz", "Double the delay time of the processsor's ALU" (cf. Fig. 1)). In nearly all cases, the questions have been answered substantially faster if VYPER! was applied (up to 15 times). This speed-up mostly results from the automatic structural analysis, the abstracting views on the code, and the immediate access to the relevant VHDL code fragments via the hierarchically organized graphical interfaces.

In the case of very small models we observed that the usage of UNIX tools like grep and texteditors (including a VHDL-specific color-highlight mode) might outperform VYPER!. This is especially true, it the models follow a few basic rules of design for reuse, like indentations, meaningful identifier names and reasonable comments. However, the use VYPER! has avoided some VHDL misinterpretations even in these cases.

9. Conclusions

This paper has presented basic concepts for the reverse engineering of HDL designs. Especially, the immediate access to interrelated design data and the abstracting graphical representations facilitate the analysis of HDL code thus promoting the reuse of former design activities. Furthermore, the analysis capabilities ease the error detection during model development. This contributes to the quality of a VHDL design which enhances its applicability for future reusage. Of course, the presented tool-set can not cover all reuse aspects of electronic design (e.g. layout macros) because it targets only HDL based design data.

In the future, we plan to integrate edit and search features into the hypertext interface. An extended design elaboration will enable the user of the tool-set to quickly answer not only analysis questions like "where is it declared?" but also "where is it used?". Finally, a more efficient practical usage of the tool-set may be achieved by integrating it into a HDL simulator in order to avoid extra compilation times.

References

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

[2] P. J. Ashenden. *The VHDL Cookbook*. University of Adelaide, Australia, 1990.

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

[4] N. Chapin. New Format for Flowcharts. *Software - Practice and Experience*, (4), Oct. 1974.

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

[6] 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.

[7] *IEEE Std 1076-1993*. Institute of Electrical and Electronics Engineers, Inc., New York, NY, USA, 1994.

[8] 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.

[9] G. W. Ledenbach. Panel: A Common Standards Roadmap. In *33rd ACM/IEEE Design Automation Conference*, Las Vegas, NV, USA, 1996.

[10] G. Lehmann, B. Wunder, and K. D. Müller-Glaser. A VHDL Reuse Workbench. In *EURO-VHDL*, Geneva, Switzerland, 1996.

[11] C. H. Lindsey. Structure Charts - A Structured Alternative to Flowcharts. *ACM SIGPLAN Notices*, (11), Nov. 1977.

[12] J. Martin and C. McClure. *Action Diagrams: Clearly Structured Program Design*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1985.

[13] J. Martin and C. McClure. *Diagramming Techniques for Analysts and Programmers*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1985.

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

[15] I. Nassi and B. Shneiderman. Flowchart Techniques for Structured Programming. *ACM SIGPLAN Notices*, (8), Aug. 1973.

[16] 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.

[17] L. L. Tripp. A Survey of Graphical Notations for Program Design - An Update. *ACM SIGSOFT*, 13(4), Oct. 1988.