VYPER!
A VHDL Hypertext Environment
for System Design Reuse

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

Abstract
Increasing complexity, tight development schedules, and small budgets characterize the present design process of electronic systems. Therefore, the reuse of design know-how and existing design data is of great importance. VYPER! supports the reuse of any arbitrary collection of VHDL source code by an automatically generated documentation environment. Hypertext links and abstract graphical design representations ease the understanding problem during the reuse of unknown designs.

1 Motivation
The increasing complexity of electronic systems and decreasing time to market intervals impose tight constraints on the current design process. Therefore, besides the evolution of design tools towards more abstract design descriptions, the reuse of design know-how and existing design data is gaining importance. Design reuse decreases development cycles and costs. It also increases design quality. Former investigations in the area of software engineering have proved that consequent reuse may reduce design expenses up to 85% [1] [2].

During the conception of VHDL, the American DoD (Department of Defense) already identified the reuse of design data as a major topic. Since electronic systems may have very long life cycles (up to 30 years), the reusable standardized language VHDL should mainly tame the long-duration maintenance problem [3].

Besides several library or model generation tools [4] [5] [6] which particularly support the reuse of VHDL design know-how, the direct exchange of VHDL source code plays a major role in system design reuse. This exchange takes place between the supplier and the customer of a design project, between cooperating project teams, during system maintenance, or simply when VHDL benchmarks are reused.

On the one hand, a basic requirement for the reusability of a VHDL design is its adaptability to the specific constraints of a design task. To this end, VHDL offers numerous language constructs, like generics, generate statements, or configurations which really simplify a 'design for reusability' [7] [8]. These capabilities allow to keep the design implementation independent of the port bitwidth, active signal levels, or subcomponent selection. In [9] it is shown that even complex components like microprocessor interfaces or DRAM controllers can be implemented as universal VHDL modules.

On the other hand, these extensive language capabilities allow very different coding styles and prevent from an easy VHDL code 'recycling'. A VHDL designer will only reuse VHDL models, if he is able to quickly realize the functionality and intention of a design [10]. Otherwise, reuse will fail because of the 'not invented here' syndrome.
2 Problem Definition

The mentioned limitation of VHDL code reuse results from the large number of language constructs, the various structuring capabilities and the complex simulation semantics. For this reason, the abbreviation VIHDL is often interpreted as 'Very Hard to Deal with Language'.

The succeeding examples illustrate some reuse limitations:

Structuring: If the following sequential signal assignment:

\[ y \leftarrow \sin(x); \]

is considered during the analysis of VHDL source code, the designer might want to know the kind and data type of object \( x \). For instance, \( x \) could be a signal, a variable, etc. Its data type might be an array of an integer or floating point type. To obtain this information, the user has to identify the accompanying object declaration, possibly a subtype declaration, and finally the data type declaration. Unfortunately, these declarations may be kept across different declarative parts, like block, architecture, entity, or package declarative parts. Fig. 1 illustrates this reuse limitation.

```
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 \leftarrow \sin(x);
  ...
```

Figure 1: VHDL example illustrating the structuring problem

Additionally, the designer has to keep in mind the VHDL rules for scope and visibility during this identification 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.

Overloading: VHDL allows to overload literals, operators, or subprograms. Overloading can enhance the readability of a design or increase the application range of an operator. Supposing that the interpretation of an overloaded element is not intuitive, the designer of course has to identify the corresponding implementation among different variants. This would require a context analysis of the overloaded subprogram or operator (e.g. number of parameters).

Model execution: The execution of the concurrent statements inside a VHDL architecture is controlled by events on signals. The VHDL reference manual [11] certainly contains an unambiguous definition of simulation semantics, but a VHDL user will have problems to comprehend the execution sequence, even if small models are considered. This results from the fact, that the architecture statement part has to be read in a nonlinear manner (moving up and down), because the statements are executed in parallel. This is comparable to the understanding problems caused by the 'goto' programming style.

Therefore, the designer will only get an impression of the execution scheme of a VHDL model if the trigger conditions (signal sensitivities) of all concurrent statements are deeply analysed. Additionally, the corresponding signal drivers have to be identified.

Investigating the following piece of VHDL source code, the user might expect at a first glance that function funct_1 will be called whenever an event on signal \( a \) occurs:

```
PROCESS
BEGIN
  proc_1;
  WAIT ON a;
  b \leftarrow funct_1(a);
END PROCESS;
```

Unfortunately, this assumption is only valid if procedure proc_1 contains no further wait statements. This small example already
proves that the analysis of unknown VHDL designs might be a complex and error prone task if no further analysis tools are provided.

The enrichment of the VHDL standard (IEEE 1076-1993) and all extensions planned in the future (e.g. IEEE 1076.1) will enlarge the reusability problems which are imposed by VHDL’s complexity.

3 The VYPER! Approach

The VYPER! environment (VYPER = Vhdl hYPERtext) is aimed to attack the understanding problem of VHDL designs in order to ease the reuse of VHDL source code. Starting from an arbitrary and unsorted collection of source code files, VYPER! automatically generates abstracting views on structure and functionality of a VHDL design (Fig. 2).

The major quality of VYPER! is the generated web of hypertexts. Hyperlink methods have been successfully applied in the area of technical documentation, because technical documents typically contain a large amount of interrelated information like footnotes, cross references, indices, quotations, or links to the appendix. The corresponding hypertext graph represents pieces of information (nodes) which are connected via associations (edges). This graph allows to easily trace the nonlinear arranged pieces of information.

Since VHDL designs also include a lot of ‘cross unit’ relations and are considerably structured, hypertext techniques qualify for analysing existing VHDL designs.

3.1 VHDL Hypertext

VYPER! allows to automatically attach hypertext cross references to the source code of all VHDL design units. For example, VYPER! generates the hypertext window depicted in Fig. 3 of the design unit behave from the following piece of VHDL source code:

ENTITY pmg IS
  PORT (en, IN std_ulogic;
       state, IN state_type;
       data, INOUT std_ulogic;
       o, OUT std_ulogic);
END;

USE work.comp.ALL;

ARCHITECTURE behave OF pmg IS
  SIGNAL start, ready, clk, q:
    std_ulogic := 'X';

BEGIN
  xor 2: PROCESS
  BEGIN
    start <= en XOR data;
  END PROCESS;

  nor_inst: nor2
    PORT MAP (start, ready, q);

  ready <=
    en WHEN state /= 2
    ELSE
      'X' WHEN ((q NOR clk) OR en) = '1'
    ELSE
      '1';

  logic: PROCESS
  BEGIN
    data <= q AFTER 3 ns;
    o <= q NAND en;
    WAIT ON ready, data;
  END PROCESS;
END;

Every bold-faced identifier in Fig. 3 provides a hyperlink which points to a specific part of the VHDL source code:

- `comp` or `pmg` link to the corresponding VHDL design unit, that is, a mouse click on these identifiers opens a hypertext window depicting entity `pmg` or package `comp`.

- `start`, `ready`, `clk`, or `q` inside the signal declaration statement provide a link to the corresponding type declaration of `std_ulogic`.

- The same identifiers inside the architecture statement part point to their signal declaration. A white mark is used to highlight the
source code position of a specific identifier inside this declaration (like c1x in Fig. 3).

- A mouse click on en, state, data, or o opens the entity psg where these objects are declared as port signals. The source code positions of these identifiers inside the port declaration will be highlighted. Of course, the entity itself might contain again several hyperlinks which point to further declarations (e.g. a link from the port signal state to the package sst where state.type is declared).

- nor_inst links to the corresponding component declaration of nor2.

- Finally, the operator identifiers XOR, /=, NOR, OR, =, and NAND offer a quick access to their declaration. In general, this is particularly useful, if operator overloading is applied.

Beside these examples, nearly all other types of links from an identifier to its declaration are included at the moment (e.g. variables, generics, constants, attributes). The select bar on the right side of the hypertext window (Fig. 3) makes it possible to activate specific types of these links. For example, if the users selects the signal hyperlinks, VYPER! would search for all VHDL identifiers denoting a signal object. Hereafter, the corresponding source code positions are tagged and the links to the signal declarations are created. Therefore, VYPER! is different from the most hypertext applications because it creates a dynamic and user-defined link web. This leads to the following key benefits:

- Only required hyperlinks are displayed. The users is not confused by unnecessary information.

- The VHDL identifiers may be classified. This might be used, for example, to rapidly distinguish between internal signals and external signals (ports) inside a VHDL netlist comprising many component instantiations. Furthermore, it is possible to ascertain if and where, for example, generic constants are used inside a design unit.
USE work.comp.ALL;

ARCHITECTURE behave OF pmg IS

SIGNAL start, ready, clk, q:
std_ulogic := 'X';

BEGIN

xor 2: PROCESS
BEGIN
start <= en XOR data;
END PROCESS;

nor inst: nor2
PORT MAP (start, ready, q);

ready <-
en WHEN state /= 2
ELSE
'X' WHEN ((q NOR clk) OR en) = '1'
ELSE
'1';

logic: PROCESS
BEGIN
data <= q AFTER 3 ns;
q <= q NAND en;
WAIT ON ready, data;
END PROCESS;

END;

Figure 3: VYPER! hypertext document, automatically generated on request for VHDL design units

- The hypertext documentation is available very fast since it is not required that a complete web of links is created in advance.

Links which have been selected already are underlined (e.g. state or clk in Fig. 3). Moreover, the user may move along a previously selected path of hyperlinks (like the path shown in Fig. 1) via the two "Back" and "Forward" buttons. It is up to the user if a new hypertext window is created for each opened VHDL design unit (Display Mode "multilevel") or if the source code is displayed inside the currently active window (Display Mode "hypertext"). The third display mode ("onelevel") does not keep the selected path. This implies a small increase of performance.
3.2 VHDL Process Model Graph

In addition, the user may invoke an abstracting graphical representation of entities and architectures via the "Graphic" button. A so-called process model graph was chosen to illustrate the dynamic behavior of the concurrent statements inside an architecture (A similar graphical representation is introduced in [12]). Fig. 4 depicts the process model graph resulting from the architecture behave.

The nodes of the graph represent all concurrent statements inside one architecture, except concurrent assertions. Different kinds of statements (e.g. signal assignment, component instantiation, process, or procedure call) are distinguished by different outline colours. The edges attached to the nodes describe signals which may activate concurrent statements and which are driven by other statements. For example, signal q is driven by instance nor_inst and additionally may activate the conditional signal assignment inside behave.

Signals which do not activate any statement are neglected in order to keep the graphics simple. Furthermore, a signal is neglected if it is a member of the sensitivity set of a concurrent statement.
but has no driver inside the architecture. This is the case for the internal signal `clk` since it actually does not activate the conditional signal assignment of `ready`.

If a loop is attached to one node, the corresponding concurrent statement may invoke itself via at least one signal (see Fig. 4: port signal `data`). A loop without any signal identifier represents a concurrent statement that has no trigger conditions. This often happens during the modeling of combinational circuitry if the wait statement or the sensitivity list will be forgotten ('endless loop'; see Fig. 4, process `xor_2`). 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 (e.g. buried "WAIT;" in `proc_1`, see first VHDL example).

These examples demonstrate that the automatically generated process model graph not only qualifies for VHDL code reuse but also might be used for error detection inside large models with complex trigger conditions ("WAIT ON ... UNTIL ...").

All nodes and signal identifiers inside the process model graph provide a hyperlink which supports an immediate access to the corresponding source code positions inside the architecture unit. For instance, a mouse click on signal `ready` will highlight the concurrent statement which drives signal `ready`. Furthermore, the user may interactively edit the generated process model graph without destroying the mentioned hypertext capabilities.

### 3.3 Additional user interfaces

Besides the presented hypertext tool and process model graph, some more user interfaces are offered by VYPER! (Fig. 2):

- The `entity graphic` module depicts the interface (ports and generics) of VHDL entities. Several hyperlinks provide links to the source code or links to some meta information (curved lines in Fig. 2).

- The `unit manager` shows the complete hierarchy of the analysed VHDL source code collection. All possible entity/architecture pairs (in terms of the LRM: 'Design entities') and their corresponding configurations are displayed (Fig. 5). A filter function allows to identify specific units inside large designs.

- Additionally, VYPER! comprises a regular `file manager` (not shown in Fig. 2).

Any tool may be invoked from any other tool, as it is outlined by the straight lines in Fig. 2. For example, the unit manager allows to open a hypertext window for each design unit, whereas the hypertext itself provides a link back to the unit manager to indicate the current location inside the design hierarchy. Finally, it is possible to start a VHDL simulation. A simulator specific makefile is derived from the hierarchy information for this purpose.

### 4 Implementation

The implementation of VYPER! is mainly based on non-commercial tools. The graphical front-ends are constructed with the TCL/TK toolkit, versions 3.6 and 7.3 [13]. This toolkit offers sufficient features to realize editable graphics with hyperlinks attached on it. Scroll bars, zoom sliders or postscript back-ends can be implemented easily.

### 5 Conclusions and Future Work

Several application examples (R6502 processor, DP32 processor [14], etc.) have shown that the presented 'push-button' generation of a hypermedia VHDL documentation relieves from the understanding problem and simplifies VHDL source code reuse.

The performance of VYPER! is sufficient, if normal-sized design units are applied. Approximately one CPU sec. on a SUN Sparc10 workstation is consumed during the dynamic construction of a window containing 100 hyperlinks.

Our future activities will focus on some additional graphics and VHDL meta information to be generated by VYPER!. Furthermore, we plan to implement an editable hypertext source code window, including an update feature. A select bar will be added to the process model graph in order to provide user configurable displaying properties. The
search and drawing algorithms will be improved to allow the handling of very large VHDL design units containing more than 2000 hyperlinks.

Finally, we regard VYPER! as our first step towards a general VHDL design database which provides a 'hypertext oriented query language'.

Acknowledgements

The authors would like to thank Frank Heynig, Peter Elter and Hans-Jörg Bottenschein for their valuable support during the implementation process.

References


