A Platform for VHDL Visualization

Zheng Lu*, Abdulhadi Shoufan*, Giudo Rößling‡
*Intergrated Circuits and Systems Labs, TU-Darmstadt, Germany
Email: {zheng,shoufan}@iss.tu-darmstadt.de
† Dept. of Computer Science, TU-Darmstadt, Germany
Email: roessling@acm.org

***Abstract*—This paper presents a new platform for VHDL visualization to support undergraduates in learning this hardware description language. The presented platform, denoted as VISUAL-VHDL, enables students to enter their own VHDL code and control an animation process, which shows step-by-step how the different language constructs are treated to synthesize a complete digital circuit. Furthermore, VISUAL-VHDL enables the visualization of the Quine-McCluskey algorithm, which is embedded in our tools to optimize the circuit resulting from synthesizing the VHDL code.**

I. Introduction

VHDL is a well-established language for hardware description in industry and research and has gained considerable attention in education [1]–[3]. An efficient learning of VHDL relies on understanding it as a hardware description language and on a clear differentiation between this language and traditional programming languages such as C.

In particular, writing a software program mainly consists of understanding the problem, translating the specification into the language syntax, and testing the written program. Questions relating to resource allocation, mapping of tasks to resources, or scheduling are out of interest, as a rule. This is attributed to the fact that most computer systems today still include only one central processing unit —although two or more cores are common in current computers— which makes the problems of resource allocation and task mapping irrelevant during software programming. Due to the sequential operation of these CPUs, furthermore, the scheduling task is also trivial.

Using VHDL to model digital systems, in contrast, is highly different. Besides understanding the system specification, knowing the language syntax, and the need to test models by simulation, VHDL designers are responsible for solving all the above problems of allocation, mapping, and scheduling, along the way—or perhaps essentially. Understanding VHDL relies on the awareness that writing any VHDL statement may be associated with allocating a new hardware resource, mapping some task of the system specification to this resource, and scheduling the execution of this task at a specific time point or in a specific clock cycle.

To make students familiar with its basic concepts, we created a visualization and animation platform for VHDL. With the aid of these tools, denoted as VISUAL-VHDL, students can enter small VHDL codes and control an animation process to see how this code is processed to set up the corresponding digital circuit compounded of basic logical elements such as gates, flip-flops and multiplexers.

TABLE I
Commercial Schematic Viewers vs. VISUAL-VHDL

Commercial Schematic Viewers	VISUAL-VHDL
Schematic is generated after completing the synthesis process which often takes a considerable amount of time.	Schematic is generated on the fly.
Schematic is output all at once. Understanding which VHDL statements were mapped to which schematic elements often demands an accurate investigation of the schematic and the VHDL code.	Schematic is generated dynamically in an interactive mode. During this animation, relating VHDL statements and schematic elements are highlighted using colors.
Investigation of relation between the VHDL code and the schematic often demands a switching between the windows of the VHDL editor and the schematic viewer.	Both the VHDL code and the schematic are displayed on one window which considerably facilitates analysis.
Code optimization may hinder the understanding of the mapping process of the VHDL code to hardware elements.	Code optimization is done on demand. User can switch on or off the optimization option.
Code optimization is performed in the background.	Code optimization can be visualized in an auxiliary window.

VISUAL-VHDL differs from schematic viewers embedded in most commercial synthesis programs in several points, which spring from the educational merit of VISUAL-VHDL, as shown in Table I.

Animation platforms for data structures and algorithms in terms of pseudo code or software programs have long been used for educational purposes and evaluated for effectiveness [4], [5]. To our knowledge, neither VHDL nor other hardware description languages were addressed in the scope of such visualization environments, so far. VISUAL-VHDL is a first step in this direction.

The remainder of the paper is structured as follows. Section II provides a brief introduction into ANIMAL, which our platform is based on. Section III details VISUAL-VHDL. Section IV concludes the paper with a summary and an outlook.

II. ANIMAL

VISUAL-VHDL is a plug-in for ANIMAL, which in turn is a Java-based environment for algorithm visualization [6]. The animation is created by applying appropriate effects to pre-defined graphical primitives such as points, polylines,

Listing 1. Example for ANIMALSCRIPT

```
1 triangle ''d1'' (25,100) (25,110) (55,110)
2 polyline ''p0'' (35,0) (35,20) (85,20)
      (85,90) hidden
3 move ''d1'' via ''p0'' within 3000 ms
```

polygons, arcs and texts. For each primitive, several specific properties such as the size and the color may be defined. The animation effects include display, timed display, hiding, color change, movement and rotation. Animations are displayed with video player-like functionality including play, pause and a direct jump to a given step. ANIMAL takes as input a special ASCII-based script denoted as ANIMALSCRIPT, which defines the animation content in a flexible way [7]. Each line in ANIMALSCRIPT can represent a command compounded of a keyword and a number of parameters. Listing 1 depicts a section of an ANIMALSCRIPT file. In this section, a triangle is first displayed. Then a hidden polyline is specified. The triangle is finally moved along the polyline during 3 seconds.

III. VISUAL-VHDL

VISUAL-VHDL extends both the graphical library of ANIMAL and ANIMALSCRIPT. Figure 1 shows the general flow for generating an appropriate animation for a given VHDL code. During its analysis, the VHDL code can optionally be optimized on the Boolean level based on the Quine-McCluskey algorithm [8]. If desired, this optimization process can also be visualized step-by-step.

The next task is to generate an *extended netlist*, which is a structural description of the digital circuit enhanced with visualization and animation information. This information is generated in the style of ANIMALSCRIPT, so that it can be treated by ANIMAL. The circuit primitives of the extended netlist are selected from a graphical library, which was extended with new classes to support digital logic schematic. Besides the automatic approach, VISUAL-VHDL allows the generation of an extended netlist from a schematic editor.

The core functionality of VISUAL-VHDL consists in interpreting the extended netlist and constructing an animation

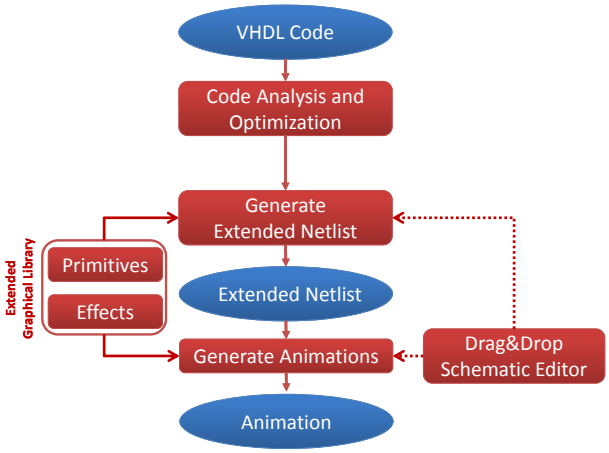

Fig. 1. Generation of an Animation for VHDL Models

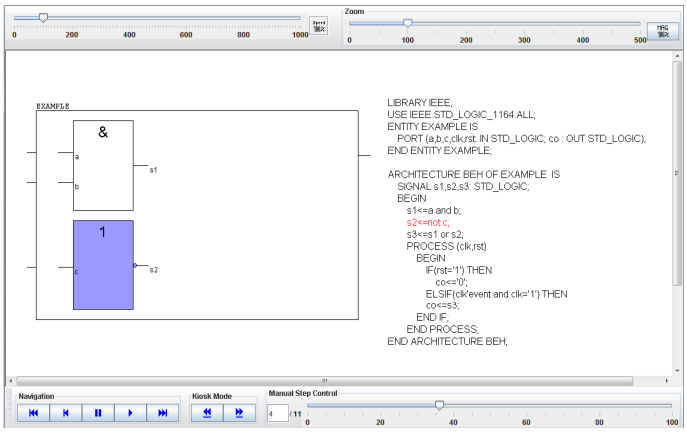

Fig. 2. Example for VHDL Animation, Processing the Assignment of the Signal S2

that can be viewed inside ANIMAL, utilizing the Java Swing library. This animation program can then be executed under user interaction to generate the circuit schematic corresponding to the analyzed VHDL code step by step. The user interaction is performed within the animation window with video player-like controls.

Figure 2 presents a screenshot of the animation window in an early step of the visualization process. Note how VISUAL-VHDL highlights the code row ($s2 <= not\ c$), which relates to the currently visualized digital inverter. In the final step, the animation window appears as shown in Figure 3.

The extended graphical library is based on the graphical library of ANIMAL and supplements it with new primitives to display logical gates, flip-flops, multiplexers, entities etc. The new special class *Wire* in VISUAL-VHDL is used to connect the terminals of different primitives.

In the following, we describe some important aspects of VISUAL-VHDL in more detail.

A. Code Analysis and Optimization

In its current prototype, VISUAL-VHDL supports the following VHDL language constructs: entities, ports, signals,

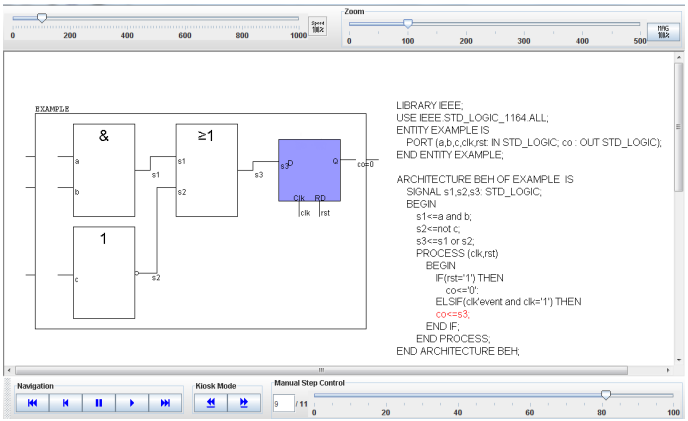

Fig. 3. Example for VHDL Animation (Last Animation Step)

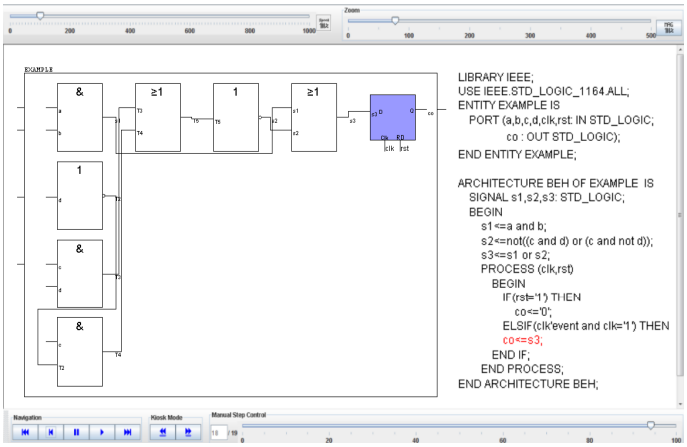

Fig. 4. Example for VHDL Animation (without Optimization)

architectures, processes, concurrent and sequential signal assignments, variables and variable assignments, conditional assignments, component declarations and instantiations.

From a design perspective, VISUAL-VHDL allows the visualization of VHDL models which presents behavioral, structural, and mixed models, i.e., models with both behavioral statements and component instantiation. Regarding behavioral models, both combinatorial and sequential logic are supported, so that a description on the register transfer level (RTL) is allowed. Description on the RTL level is the most well-known approach to specify hardware in commercial design flows. On this level, the designer takes the responsibility for bit-accurate resource allocation, task mapping, and cycle-accurate scheduling. Thus, with the aid of VISUAL-VHDL, students do not only learn VHDL and digital logic, but the de facto standard design approach on the RTL level.

Upon parsing and analyzing the VHDL model, VISUAL-VHDL offers an optimization of this model. VISUAL-VHDL currently uses the Quine-McCluskey algorithm [8] to find the minimal form of the Boolean function represented by the VHDL model. Students can switch the optimization process on and off to learn the effect of optimization on the synthesis result. Figure 4, for instance, visualizes the synthesis result of a given VHDL code without optimization. Investigating the code shows that the signal $s2$ can simply be determined by inverting the signal c as $c.d + c\bar{d} = c$. Thus, if the same code is visualized with optimization, the synthesis result would appear as given in Figure 3.

The Quine-McCluskey algorithm relies on finding prime implicants, which is an NP-hard problem. This algorithm is taught in many courses on logic design. Besides its usage during synthesizing the VHDL code, VISUAL-VHDL provides the possibility to visualize the proceeding of this algorithm. By this means, students learn about processes running in the background of the synthesis task.

B. Extended netlist

The extended netlist includes all the information needed for the dynamic visualization of the digital circuit. An extended netlist extends ANIMALSCRIPT with several primitives to visualize circuit symbols. During the animation process, an extended netlist is processed sequentially from top to bottom. Listing 2 shows the section of the extended netlist responsible for visualizing the flip-flop and its connection with the OR-gate according to Figure 3. In particular, this script section contains two animation steps parenthesized with curly bracket {}:

Listing 2. An Extended Netlist Section Relating to Figure 3

```
1 {color ''or1'' type ''fillColor'' none
2  d ''d1'' (421,117) (571,267) input ''s3''
       output''co'' clock''clk'' reset''rst''
       color black fillColor (153,153,255)
       depth 50
3  unhighlightCode on ''codeSource'' line 16
4  highlightCode on ''codeSource'' line 17
5 }
6 {wire wire_or1-0->d1-0 (404,192) (404,178)
      (421,178)
7 }
```

1) The first step performs the following four actions simultaneously. (1) The fillcolor of the OR-gate is removed. (2) A D-flipflop is visualized, with upper-left and lower-right corners at the position (421,117) and (571,267), respectively. Note that the y-axis in VISUAL-VHDL is directed downwards. The names of the inputs and outputs of the flip-flop are specified. The attribute *color* specifies the color of the flip-flop frame and the signal names. The flip-flop is finally highlighted by a fillcolor. (3) Line 16 in the VHDL code, which was highlighted in the previous animation step, is unhighlighted. (4) Line 17 ($c0 <= s3$) is highlighted.
2) In the second step, a wire from the output of the OR-gate to the input of the D-flipflop is visualized. Note that this wire is specified by three points, as depicted in Figure 5 schematically. See Figure 3 for comparison.

The automatic generation of an extended netlist for a given VHDL code is a highly complex task, which includes the following subtasks:

1) Determining the optimal placement of the logical elements.
2) Determining the optimal routing.
3) Determining the optimal animation.

The execution of the first two subtasks results among others in the (x, y) position data for all the elements and wires of the circuit. These data are supplied as parameters in the extended

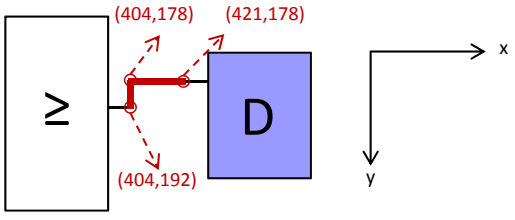

Fig. 5. Illustrating the Wire Dimensioning in the Extended

netlist, see Listing 2.

As mentioned in Section III, an extended netlist is generated at the end of the analysis and optimization phase. For this purpose, each element is specified by an attribute, which gives the number of the VHDL code line relating to that element. This attribute is required during the construction of the extended netlist to highlight the VHDL lines and the corresponding circuit elements synchronously.

For an appropriate display of the digital circuit, the animation field of the animation window is organized as a grid of numbered cells and channels. Logical elements are placed into cells, while wires are laid within the channels.

The grid size in terms of cell number and size is automatically adjusted according to the netlist content. Some rules are defined for simplifying the placement process. One rule relates to the placement of the circuit elements having external interface. Elements with external input signals, for instance, are placed in the leftmost grid column as far as possible. In contrast, elements with external output signals are placed in the rightmost grid column.

Visualizing wires in VISUAL-VHDL is a sophisticated task for the following reasons:

1) Wires should be as short as possible.
2) Wire segments can be only horizontal or vertical, not diagonal.
3) Suitable inflection points should be found.
4) Intersections should be minimal.
5) Wires should expand from the output of an element to the input of another. The expansion velocity should be adjustable.

To provide this flexibility, our *Wire* object is realized as a set of points. To visualize a wire consisting of several segments, the start points of each segment and the end point of the last segment are provided as parameters, see Figure 5.

C. Schematic Editor

The schematic editor as shown in Figure 6 is an extension of the graphic editor of ANIMAL. The new digital toolbar at the right includes symbols for 15 element types including gates, flip-flops, multiplexers and demultiplexers. Upon dragging and dropping a symbol, several parameters can be set, such as the name and the color. The number and the names of inputs can be entered for each gate. For a multiplexer, for instance, the user may set the number of the data inputs. The number of the control signals is then determined internally to avoid errors. A D-flipflop can optionally be provided with set, reset and/or clock-enable signals.

In addition to plotting, the schematic editor enables the simulation of simple combinatorial circuits. For this purpose, students can define the digital value for each input of the plotted elements. The simulation core determines the output values and visualizes them automatically.

IV. CONCLUSION

VISUAL-VHDL is a visualization platform for VHDL. It enables entering VHDL code and an interactive production of circuit schematic. By this means, students can learn the effect

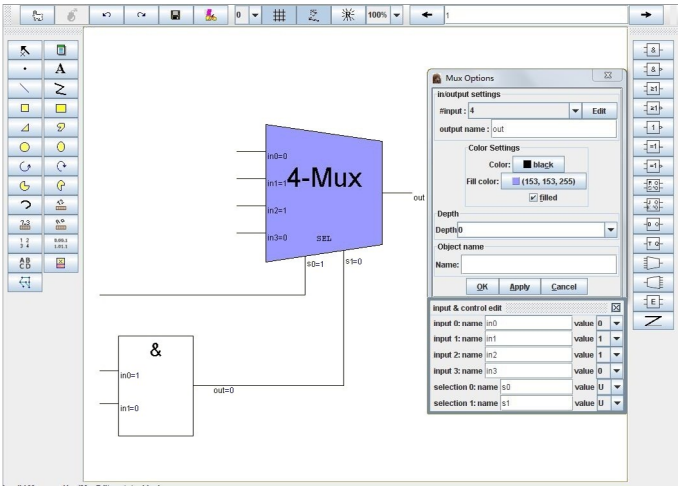

Fig. 6. VISUAL-VHDL Schematic Editor

of the versatile language constructs on the resource allocation, task mapping and scheduling in the target system. Educators may also take advantage of this tool to verify their models or to quickly generate circuit schematics using the drag&drop toolbar of the schematic editor.

VISUAL-VHDL is the first step toward a sophisticated system to support the learning process in many subjects of computer and electrical engineering. Currently we are developing a web interface for VISUAL-VHDL. Besides facilitating the usage of our tools the web interface includes a feedback system enabling students to evaluate these tools, so that we can prove their effectiveness in the near future. Furthermore, our platform will be completed and developed to support further features of VHDL and its event-driven simulation process. Other hardware description languages, such as Verilog, will also be considered.

REFERENCES

[1] A. Sagahyroon, "From AHPL to VHDL: A course in hardware description languages," *IEEE Transactions on Education*, vol. 43, no. 4, pp. 449–454, 2000.

[2] I. Ainhoa Etxebarria and M. Sanchez, "An Educational Environment For VHDL Hardware Description Language Using The WWW And Specific Workbench." *Frontiers in Education Conference*, pp. 2C 2–7, 2001.

[3] E. Gutiérrez, M. Trenas, J. Ramos, F. Corbera, and S. Romero, "A new Moodle module supporting automatic verification of VHDL-based assignments," *Computers & Education*, vol. 54, no. 2, pp. 562–577, 2009.

[4] C. Hundhausen, S. Douglas, and J. Stasko, "A meta-study of algorithm visualization effectiveness," *Journal of Visual Languages and Computing*, vol. 13, no. 3, pp. 259–290, 2002.

[5] M. Kraemer, "Balanced Cognitive Load Significantly Improves The Effectiveness Of Algorithm Animation As A Problem-solving Tool," *Journal of Visual Languages & Computing*, vol. 19, no. 5, 2008.

[6] G. Rößling, *Animal-Farm: An Extensible Framework for Algorithm Visualization*. VDM Verlag Dr. Müller, 2008.

[7] G. Rößling and P. Schroeder, "Animalipse - An Eclipse Plugin for ANIMALSCRIPT," in *Proceedings of the Fifth Program Visualization Workshop, Madrid, Spain*, G. Rößling and J. Á. Velázquez-Iturbide, Eds., 2008, pp. 95–102.

[8] G. Vastianos, "Boolean functions' minimisation software based on the Quine-McCluskey method," 2008, software Notes, (Draft Version).