# Synthesis and Simulation for Multiple Target Languages

Dirk Herrmann Rolf Ernst

Technische Universität Braunschweig
Institut für Datenverarbeitungsanlagen
Hans-Sommer-Straße 66
D–38106 Braunschweig
Germany

## Abstract

*The generation of a circuit description on register transfer level in some target hardware description language is the last step of the high-level synthesis process. We present a generalized approach to the generation of register transfer descriptions that allows to support multiple target languages in a uniform way, including a subsequent cycle true simulation of the generated circuit. This is achieved by introduction of an intermediate abstraction layer between the data structures of the high-level synthesis system and the syntax of the target languages and the use of a common simulation output format. We present our results for the target languages VHDL, Verilog and C++.*

## 1. Introduction

High-level synthesis is the transformation of a behavioral description of an application into a register transfer (*RT*) level description [4]. The RT description is generated in some target hardware description language that can be further processed by RT synthesis tools like Synopsys. To suit the need of different circuit designers commercial synthesis systems have to support several target languages. However, for the developer of the synthesis systems the support of each additional target language means increased development and maintenance costs. To our knowledge, the efficient generation of RT descriptions (*net lists*) for multiple target languages has not been investigated yet.

In this paper we present our approach to the final RT code (or net list) generation phase of the high-level synthesis process and a subsequent cycle true simulation. By introduction of an intermediate abstraction layer during net list generation our approach allows to generate code for multiple target languages generically without changing the quality of the generated circuit. The program code that is specific for each target language only requires about one fifth of the code lines of the generic routines which are common for all target languages, which reduces the maintenance cost for the high-level synthesis system. Due to the large percentage of shared code the organization and style of the generated code is similar for all target languages, which makes debugging easier.

The generated RT descriptions contain output directives that report the system state during simulation but can be ignored for RT synthesis. A Lisp style output format is used to report the simulation state. The simulation output is postprocessed to achieve a cycle based visualization of the simulation run and the results. This representation is identical for all target languages since the postprocessing compensates differences between different target languages like varying process execution orders between VHDL and Verilog. Since the simulation stimuli are also generated from a common input format, the whole process of code generation and simulation is target language and simulator independent.

The RT net list generation was implemented in C++, with output functions for VHDL [3], Verilog [11], and C++ [10]. For C++ a dedicated simulation library was provided that allows to model parallel execution of processes. The postprocessing of the simulation output is done in Scheme, a language from the Lisp family of programming languages. Due to the Lisp style simulation output format the Scheme postprocessing code can be kept extremely simple and allows to check for arbitrary conditions during simulation, which makes debugging fairly comfortable.

In the next chapter we will give an overview of the overall design flow in our synthesis system. Section 3 describes how code generation was realized in a uniform way for the different target languages. In section 4 we present how language and simulator independent simulation is achieved. Finally some results show the effectiveness of our approach before we conclude the paper.

This work is supported by the Deutsche Forschungsgemeinschaft (DFG)

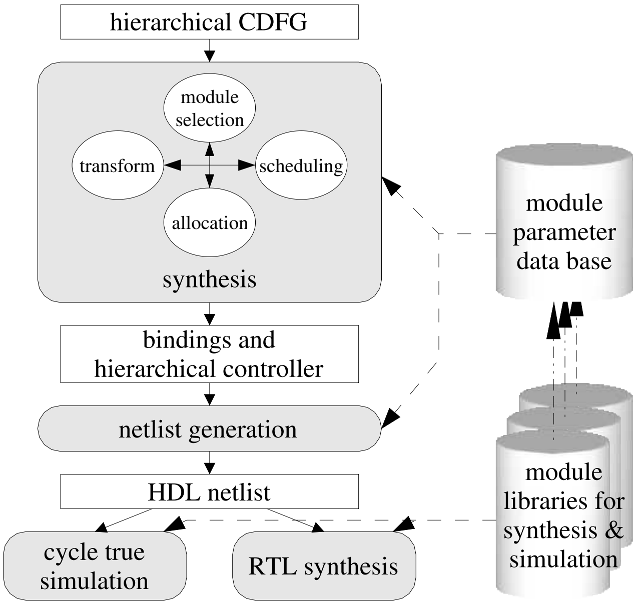


**Figure 1. Synthesis Design Flow**

## 2. Design Flow

Figure 1 shows the design flow of the Braunschweig Synthese System (*BSS*), which we used for our experiments [5]. An input description is given as a hierarchical control and data flow graph (*CDFG*), i. e. every node of some CDFG within the hierarchy is either an operation or represents another CDFG. The hierarchy is used to express control structures like nested loops or functions calls. On the input CDFG the synthesis subtasks code transformation, functional unit type selection, scheduling and allocation are performed iteratively. The results of the synthesis process are a mapping of operations to functional units (*FUs*) and a hierarchical controller that reflects the structure of the CDFG after all code transformations have been applied.

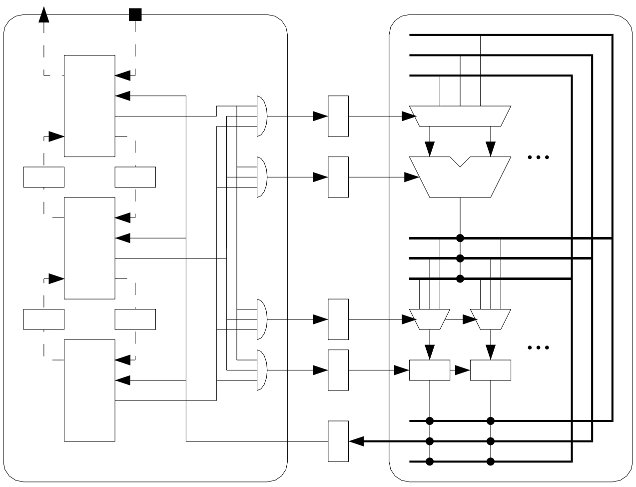


**Figure 2. Synthesis Result**

The register transfer structure of the generated circuit as it is defined by the controller hierarchy and the binding of operations to FUs is shown in figure 2 with the control path on the left and the data path on the right. The control path consists of a set of sub-controllers that correspond to the controller hierarchy. The output signals of the control path are derived from the output signals of the individual sub-controllers using a simple tree of combinational logic. All input and output signals of the control path as well as the signals for communication between the sub-controllers are latched to reduce the critical path. In the data path, FUs and registers are connected by an application specific interconnect network, which may also contain chained FUs or shift registers.

All synthesis steps and the subsequent net list generation use a common data base which holds information about available types of FUs. For each type of FU the data base describes the set of available operations, execution times, area requirements and how the input, output, inout and control ports have to be connected. Thus, the data base represents only model parameters and interface information in a way that is independent of the target language and the actual implementation of the FU. For a subsequent RT synthesis and simulation, however, libraries with implementations of the FUs for the different target languages are required.

## 3. Netlist Generation

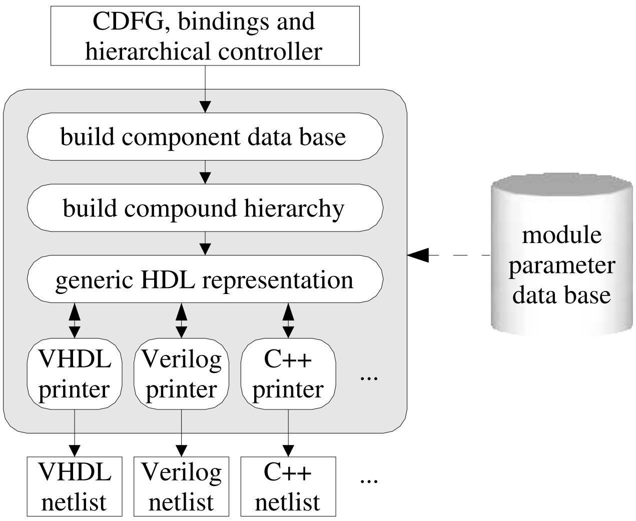


**Figure 3. Netlist Generation Design Flow**

Figure 3 shows the different phases of the net list generation process. In the first phase, the components that have to be generated, like instances of FUs, registers, multiplexors and controllers, are collected in a data base.

In the second phase, the components are compound hierarchically, which, in contrast to a flat design, allows an incremental bottom up RT synthesis. The component hierarchy can be arbitrarily chosen. For example, parts of the control and data path can be combined into a separate component if this seems to be advantageous. An extra hierarchy level is provided to combine the generated circuit with the simulation environment, e. g. external memories and clock generation.

The third phase is a preparation phase for the actual output. For each component the fine grained structure of the processes and the logic expressions is defined in an internal format that is an abstraction from the target languages. The subsequent final printing phase can easily transform this internal format into the syntax of the selected target language.

## 3.1. Target Language Abstraction

After the second phase of the net list generation there exists an internal representation that holds all necessary information to generate a circuit in any RT description language. However, most information is represented implicitly in form of internal data structures of the high-level synthesis system. The conditions for state progression in the different controllers for example are defined by the structure of the control graphs. By introduction of an intermediate layer of abstraction between the implicit circuit representation and the representation in the syntax of some target language we were able to get rid of most of the target language specific code in favor of generic routines.

Focusing on VHDL and Verilog as target languages, our internal representation allows to use signals and registers of different bit widths, `if-then-else` and `case` control structures and boolean expressions with operators like AND, OR, NOT, comparison, concatenation and bit selection. Computations are performed in parallel processes that are initiated by events, i. e. changes of signal values. Hierarchical modules allow to divide the circuit into separately synthesizable components.

This intermediate format allows for a simple transformation into VHDL and Verilog, generating RT descriptions on a high level of abstraction. The state progression of the subcontrollers for example is represented as a `case` statement. For the generation of the intermediate code as well as for the translation into the target language we made sure that the quality of the synthesized circuit after RT synthesis (in our case by using Synopsys) remained the same.

In addition to the code generation for VHDL and Verilog we implemented the code generation for C++ with the objective to allow fast simulation runs. To guarantee the same cycle based simulation behaviour we provided a run time library for C++ with data structures for processes and signals. In principle, we also could easily support languages like hardwareC [7] or TH [8]. However, our internal representation is not well suited for output formats like SLIF (combinatorial logic) or KISS (tabular automaton representation) since these languages do not provide control structures and only allow operations on one-bit signals.

The intermediate format allowed to unify large parts of the output generation. Only the tranformation of this internal format into the target language required language specific routines. These specialized routines cover syntactic differences between the languages as well as certain conceptual dissimilarities. In VHDL and Verilog, for example, the interconnect structure is generated implicitly by the declaration and use of signal names. In contrast, for C++ these connections have to be generated explicitly. However, as will be described below, the language specific code parts typically consisted only of small code pieces.

### 3.1.1. Signal Naming Convention

Signal names are generated by a common algorithm. A target language specific postprocessing of each name guarantees, that the names do not form keywords or contain illegal characters. The definition of a consistent signal naming convention is essential to guarantee the maintainability of the generated RT descriptions. Thus we use a system that allows to determine the function of a signal within the circuit from the signal name. Control signals are named by the component they control and data signals by the source component. Additional information about the way the signal is connected in the current component is added, e. g. whether it is an incoming, outgoing or local signal.

### 3.1.2. Expressions

Signal assignments or conditions for state transitions correspond to boolean expressions in our intermediate format. These expressions are represented as trees, similar to *parse trees* used for compilers [2]. Leaf nodes are constants or signals, all other nodes are operations like comparison, logical of bit wise AND, OR or NOT. Due to the tree structure it is not necessary to define operator precedences. During the output of the expressions parentheses are used to guarantee the desired operator precedence.

```
printLogicOr(Lang lg, Node l, Node r) {
    printLeftParen(lg);
    printNode(lg, l);
    if ((lg == CPP) || (lg == VERILOG))
        print(" || ");
    else if (lg == VHDL)
        print(" or ");
    else illegalLanguageError();
    printNode(lg, r);
    printRightParen(lg);
}
```

**Figure 4. Recursive Expression Output**

The translation into the target language is done recursively. A simplified example in figure 4 shows how a logical OR is printed. In the example the function `printNode` prints the node parameter by branching to the output function that corresponds to the type of the node parameter. The example shows how little language dependent code is required to support the different syntaxes. With this approach, expressions like `(((state == 1) or (state == 5)) and ((s1 = 0) or ((s1 = 2) and (s2 = 0))))` that appear in the controller code can be generated from a central function that is used for all target languages.

### 3.1.3. Control Structures

Only a limited set of control structures like conditional statements or processes is required to generate RT descriptions. Loop statements, for example, are not necessary. With the possibility to handle expressions generically, conditional statements are easily implemented: One expression forms the condition to be tested and the `then` and `else` cases are sequences of assignments.

Processes required more language dependent code pieces due to the bigger syntactic differences. Still, certain language independent tasks like the generation and printing of sensitivity lists were implemented as generic routines. Continuous assignments are a convenient short form of processes with an implicit sensitivity list in VHDL and Verilog that consist only of a single signal assignment. In C++, however, these had to be represented as ordinary processes with explicit sensitivity lists.

### 3.1.4. Modules

Every component that is generated (controllers, multiplexors, registers, FUs and compounds) is defined as a module of the target language in a separate output file. Although VHDL and Verilog both provide a module concept there are some differences: VHDL requires a separation between interface declaration (`entity`) and the implementation (`architecture`), while this is not necessary in Verilog. Due to such differences only small parts of the module code generation could be done generically. This were for example the collection and printing of the module interface parameter lists.

The instantiation of modules from within other modules, however, requires mainly the output of the parameter list. For FUs (which are implemented in an external library) the corresponding information is provided in the data base that was described in section 2.

### 3.1.5. Simulation Directives

In order to guarantee a close correspondence between simulated and synthesized circuits there is no distinction between net list generation for synthesis and simulation. Simulation directives are always generated in the RT description as output functions that can be ignored by the RT synthesis tools without changing the behaviour of the circuit. The information that is printed by the components during simulation is similar for all target languages.

## 4. Simulation

The simulation of the RT descriptions is performed with simulators for the corresponding target language, as

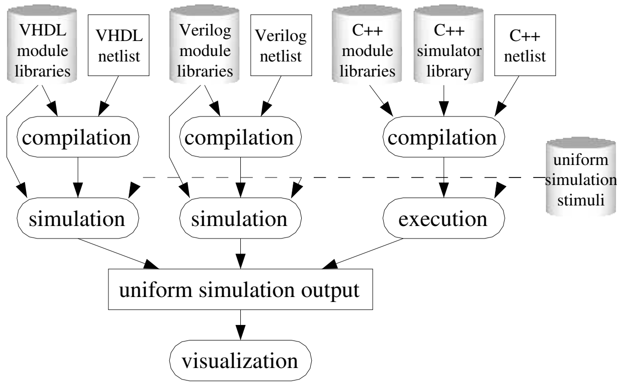


**Figure 5. Simulation Flow**

is shown in figure 5. Simulation stimuli are given in a uniform, Lisp style format which is automatically transformed into the suitable format of each simulator. The RT descriptions report the simulation state in a generic, Lisp style output format. A common postprocessor is used to visualize the sequence of simulation events and the simulation results.

For each target language the run time libraries providing the implementation of the FU types are the only target language dependent supplementary files that have to be provided for the simulation. In addition, C++ requires the run time library with the process, register and signal data types and the event based simulator core routine. During compilation of the C++ net list these libraries are linked into a single executable file.

## 4.1. Generic Simulation

In order to allow unified cycle true circuit simulation independent of the actual simulator or target language we defined a generic Lisp style simulation output format. During simulation, the full system state (clock cycles, state transitions, signal value changes, process executions etc.) is reported. FUs from the run time or simulation library have to report their internal state accordingly. This way, a common set of postprocessing tools can be used to analyze the simulation output of different simulators. Further, the correspondence between simulation events and the operations of the original input CDFG can be expressed explicitly. This can not be easily achieved otherwise, since any RT level component can correspond to a set of operations in the input description.

The Lisp style simulation output as shown in figure 6 was chosen due to its simplicity and to avoid the definition of a dedicated simulation output language. Lisp (or in our case Scheme) allows to read parenthesized expressions with a single command, thus eliminating the need for a special parser. The simulation output can be processed with any Lisp or Scheme interpreter, which allows to treat the simulation output as a data structure and analyse it with the possibilities of a full programming language.

The postprocessing of the simulation output is neces-

```
...
(cycle 12)
(state-change Block_7 "S1")
(module-in "NOT_0" ((sig "I" 0)))
(module-out "NOT_0" ((sig "O" 1)))
(mux-in "MUX_5" ((sig "NOT_0_O" 1)))
(mux-out "MUX_5" ((sig "REG_0_I" 1)))
(module-in "REG_0" ((sig "I" 1)
                    (sig "WR" 1)))
...
```

**Figure 6. Simulation Output Example**

sary for three reasons. First, the process execution order of VHDL and Verilog is not deterministic. This means, that the sequence of reported events can be quite different between these languages, between simulation runs on different simulators for the same language or even between different simulation runs on the same simulator. Second, due to glitches several assignments to the same signal can be reported in a single simulated cycle. To verify the functional correctness, however, it is only necessary to know about the value that is valid at the beginning of the next clock cycle. Third, since we use a pipelined controller structure related events from the control path and the data path are reported in different clock cycles. In the visualized simulation output, however, we prefer that corresponding events are grouped together for improved readability of the simulation results. Thus, a (configurable) reordering of the simulation events is done in the postprocessing phase prior to visualization.

## 5. Results

| | #lines | % lines | #func. | $\frac{\#lines}{\#func.}$ |
|---|---|---|---|---|
| total | 4498 | 100.00 | 497 | 9.1 |
| generic | 2941 | 65.38 | 277 | 10.6 |
| VHDL | 587 | 13.05 | 75 | 7.8 |
| Verilog | 429 | 9.54 | 73 | 5.9 |
| C++ | 541 | 12.03 | 72 | 7.5 |

**Table 1. Code Line and Function Counts**

Table 1 shows the numbers of code lines and functions of the implementation of the net list generation. The line numbers do not contain empty lines, comment lines and lines only consisting of opening or closing braces. The line 'total' shows the total line and function count, including the code for the different target languages. The other lines show the absolute and relative shares of the generic code and the specialized code for each of the target languages. The amount of specialized code that is required to provide support for each language is only about one seventh (Verilog) to one fifth (VHDL) of the generic code, which proves the benefit and efficiency of the target language abstraction.

As indicated by figure 4, many of the specialization functions are trivial. This is confirmed by the fact that the average number of lines per specialized function (including the lines for the function header) lies in the range from 5.9 for Verilog and 7.8 for VHDL, while the average length of the generic functions is 10.6 lines. Since for each target language some dedicated helper functions were defined, the number of functions differs slightly. Apart from that, most functions could either be implemented generically or required a specialized implementation for each target language. The differences in code lines for the different languages reveal the different complexities of the syntaxes of VHDL, Verilog and C++ [9, 6].

For C++ the complexity of the code generation could be kept small since the additional run time library was developed with the additional objective to allow for simple code generation. Thus the fact that C++ is a general purpose programming language rather than a hardware description language had no noticeable effect on the implementation complexity. The need to generate signal connections explicitly (see section 3.1) for example could be covered by a single specialized function.

| | VHDL | Verilog | C++ |
|---|---|---|---|
| bsearch | 7191 | 6360 | 11359 |
| bsort | 8057 | 6762 | 12362 |
| diffeq | 8052 | 7121 | 12468 |
| gcd | 5907 | 5072 | 9108 |
| ellipf | 6977 | 6406 | 10582 |
| max | 5766 | 5145 | 8832 |
| filt | 7588 | 6632 | 11893 |

**Table 2. Size of the Generated Code**

The size of the generated RT code for a set of benchmarks is shown in table 2. The first two examples `bsearch` and `bsort` are implementations of binary search and bubble sort. The algorithms `diffeq`, `gcd` and `ellipf` are taken from the HLSynth92 collection [1]. The `max` example determines the maximum element of an array and `filt` contains the inner loops of an FIR image filter.

While the different treatment of signal connections in C++ had no big influence on the implementation complexity it becomes evident in the size of the generated code. Since this effects the code according to the number of signal connections it results in a much larger code size compared to VHDL and Verilog.

| | VHDL | Verilog | C++ |
|---|---|---|---|
| bsearch | 2.29 | 0.89 | 100.68 |
| bsort | 2.42 | 0.93 | 121.18 |
| diffeq | 2.49 | 1.01 | 118.91 |
| gcd | 1.64 | 0.71 | 86.70 |
| ellipf | 2.29 | 0.81 | 96.42 |
| max | 1.89 | 0.64 | 77.01 |
| filt | 2.34 | 0.83 | 102.81 |

**Table 3. Compilation Time**

For the same set of benchmarks we measured the time for the compilation of the generated RT code (see table 3). For VHDL we used the Synopsys'97 suite on a 300Mhz UltraSparc2. C++ was compiled with the egcs-1.1.1 compiler from the Free Software Foundation on the same machine. For Verilog we used a simulator from Gateway Design Automation on a Sparc10. Despite the slower machine, the Verilog compilation was fastest. The compilation of C++ takes extremely long due to the complexity of the language and the nontypical structure of the automatically generated code. Obviously these numbers depend on the set of tools that are used.

| | VHDL | Verilog | C++ |
|---|---|---|---|
| bsearch | 6013 | 5553 | 6758 |
| bsort | 27320 | 22207 | 27452 |
| diffeq | 3940 | 3143 | 3523 |
| gcd | 3384 | 2751 | 3331 |
| ellipf | 1962 | 1620 | 2123 |
| max | 5051 | 4133 | 4819 |
| filt | 9241 | 7676 | 8906 |

**Table 4. Simulation Events**

Our C++ simulator starts the execution of a process whenever one of the events in the process' sensitivity list occurs. If more than one event occurs the process is executed several times according to the number of events. We do not perform any optimizations to reduce the number of process activations.

Table 4 shows the number of simulation events that are reported during one simulation run of the synthesized circuits in each of the target languages. The different behaviour of the simulators proves the need for the postprocessing of the simulator output to obtain comparable results between different target languages.

| | VHDL | Verilog | C++ |
|---|---|---|---|
| bsearch | 9.7 | 11.0 | 86.3 |
| bsort | 17.0 | 24.9 | 109.0 |
| diffeq | 7.5 | 8.0 | 72.7 |
| gcd | 10.8 | 10.8 | 102.8 |
| ellipf | 6.0 | 5.4 | 69.2 |
| max | 15.1 | 16.4 | 132.8 |
| filt | 13.4 | 16.1 | 101.8 |

**Table 5. Simulated Cycles/Second**

Finally, in table 5 the number of simulated cycles per second is shown. Despite the slow Sparc10, Verilog does quite well. The performance of the C++ simulator appears comparatively bad since the compiled C++ code is only about 10 times faster than the Verilog simulator, which executes on a slower machine. An analysis revealed that a lot of time consuming type conversions are performed at run time, which could be avoided with a little additional coding effort in the design of the process and signal data types.

## 6. Conclusion

We have presented a general approach to support the generation of RT net list code for multiple target HDLs. Only small amounts of specialized code have to be provided for each target language, which reduces the maintenance efforts for the high-level synthesis system.

Our results showed that support for Verilog required the least effort compared to VHDL and C++. By implementing support for C++ as a target language we have demonstrated the versatility of the approach. However, this required to provide an additional run time library and a simulation environment.

The common simulation output format allows for efficient language independent analysis of simulation results and offers the possibility for cross-language validation. During our simulation runs several bugs in the implementation of the FUs were detected this way. The use of Lisp-like input and output formats saved us a lot of coding effort since no specialized parsers were required.

In our experiments the tools for Verilog showed to be most efficient compared to VHDL and C++. The C++ results indicated that our own simulator needs some performance improvements. However, even if the C++ simulation performance would be improved, an effective use of C++ as a simulation language is only reasonable for long simulations runs due to the extremely long compilation times.

## References


[1] HLSynth'92. http://www.cbl.ncsu.edu, 1992.

[2] A. W. Aho, R. Sethi, and J. D. Ullmann. *COMPILERS Principles, Techniques and Tools*. Addison Wesley, Massachusetts, 1986.

[3] B. Cohen. *VHDL Coding Styles and Methodologies*. Kluwer, 1995.

[4] Daniel D. Gajski, Nikil D. Dutt, A. C-H Wu, and S. Y-L Lin. *High-Level Synthesis — Introduction to Chip and System Design*. Kluwer, 1992.

[5] U. Holtmann. High–Level Synthese mit BSS für den Einsatz im Hardware/Software Codesign (high–level synthesis with BSS for usage in hardware/software codesign). In *6. E.I.S Workshop*, page 208ff, 1993.

[6] O. Levia, S. Maginot, and J. Rouillard. Lessons in language design: Cost/benefit analysis of VHDL features. In *DAC*, pages 447–453, 1994.

[7] G. De Micheli, D. Ku, F. Mailhot, and T. Truong. The olympus synthesis system. *IEEE Design & Test of Computers*, pages 37–53, October 1990.

[8] T.-T. Nguyen. THUD. http://www.glug.org/people/ttn/software/thud.

[9] D. J. Smith. VHDL & Verilog compared & contrasted — plus modeled example written in VHDL, Verilog and C. In *DAC*, pages 771–776, 1996.

[10] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, 2nd edition, 1993.

[11] D. E. Thomas and P. R. Moorby. *The Verilog Hardware Description Language*. Kluwer, June 1996.