Defining an enhanced RTL Semantics

Shuqing Zhao, Daniel D. Gajski
Center for Embedded Computer Systems
University of California, Irvine
Irvine, California, 92697 USA
{szhao, gajski}@uci.edu

Abstract

In this paper we formally define an enhanced RTL semantics. This is intended to elevate the RTL design abstraction level and help bridge the HDL semantic gap among synthesis, simulation and formal verification tools. We define the enhanced semantics based on a new RTL++ language that supports pipelined operations using a new pipelined register variable concept. The execution semantics of RTL++ is specified in a structural operational semantics style aimed to form the basis for related simulation and formal verification algorithm development. A RFSM model is defined to support natively the synthesis semantics of RTL++. We also present an example of extending SystemC to support the notion of pipelined register variable.

1. Introduction

In today's SoC era most of the new hardware block designs are still conducted at the register transfer level. Engineers first use hardware design language like Verilog or VHDL to model the RTL hardware components: control logic and datapath units such as multiplexers, registers, arithmetic units etc. Synthesis, simulation, and optionally formal verification tools are then used on this model to generate the hardware logic and verify its functional correctness. Often times this is a long and error-prone process that has been and continues to be the major bottleneck of SoC design productivity. One of the main source of the problem is always attributed to Verilog or VHDL. They were designed and well suited to support gate-level simulation details e.g. their delta cycle semantics guarantee that they can simulate the combinational logic behavior even if there are combinational feedback loop within the circuit. They do not support conveniently the true RTL semantics for synthesis, simulation and formal verification. Furthermore their semantics are not formally defined and even the IEEE standard documents have ambiguities which led to incompatible simulation results among different vendors. In practice whatever coding style the dominant EDA tool accepts became the *de facto* standard for RTL semantics.

To alleviate this problem, many languages [2, 10, 6, 3] and associated tools have been proposed and implemented in the industry and academia. However in our opinion some of these languages are not suited for RTL, some inherited the exact problems that exists in today's HDLs, and some are headed in a wrong direction [4]. We believe a study on the appropriate RTL semantics is necessary and could shed light on the design of the next generation HDLs by allowing precise specifications of intended program behavior and permitting proofs that a program does (or perhaps does not) meet its specification.

In Section 2 we review some of the related works that have been accomplished. We propose a new RTL++ language as our basis to discuss and describe the enhanced RTL semantics in Section 3. In Section 4 we present the formal execution semantics for RTL++ using structural operational semantics(SOS) [12] as the framework. In Section 5 we give the definition of a new model of computation called Register-transfer Finite State Machine (RFSM) as a platform to describe the synthesis semantics for RTL++. Section 6 shows an example to extend SystemC to support the key semantic enhancement. Section 7 concludes the paper and suggests the future research direction.

2. Related Work

There has been some effort attempted to define formal semantics for existing HDLs. The most relevant one is [7] in which Gordon defined a trace semantics model for Verilog synthesizable subset. Accellera made an effort in 2001 to standardize on RTL semantics [1]. [14] provides an implementation using a C++ library to model this proposed semantics. Zhu in [15] defined a meta RTL language to enrich RTL data types. Another inspiration source has been from synchronous programming community who pioneered

1530-1591/05 $20.00 © 2005 IEEE

the synchrony concept which is also the assumption used in this paper. Esterel [3] is ideal for control-dominant application and has been adopted in control software domain while Lustre [8] has been largely applicable to digital signal processing software development. We believe a good RTL language should be simpler and more attuned for both control and data oriented application.

3. The RTL++ language

```
module diffeq (
         input wire bool start,
         output pregister<2> bool ready,
         input wire bool reset,
         input wire integer Xinp,
         input wire integer Yinp,
         input wire integer Uinp,
         input wire integer Ap,
         input wire integer DXp,
         output register integer Xoutp,
         output register integer Youtp,
         output register integer Uoutp,
         clock clk);

register integer x,y,u,a,dx;
register integer t1,t3,t4,t5;
pregister<2> integer t2; //path to t2 is pipelined

while true {
    wait until (start == 1);
    ready = 0;
    x = Xinp; y = Yinp; u = Uinp;
    a = Ap; dx = DXp;
    wait;
    while (x < a) {
      t1 = u * dx;
      t2 = 3 * x;        //start 1st pipelined op
      wait;
      t2 = 3 * y;        //start 2nd pipelined op
      x = x + dx;
      wait;
      t4 = t1 * t2;      //t2 from 1st pipelined op
      y = y + t1;
      wait;
      t5 = dx * t2;      //t2 from 2nd pipelined op
      wait;
      u = u - t4;
      wait;
      u = u - t5;
      wait;
    }

    Xoutp = x; Youtp = y; Uoutp = u; ready = 1;
}

endmodule
```

Listing 1. RTL++ code example for differential equation $y'' + 3xy' + 3y = 0$ solver

Our goal is to provide a formal semantic framework to guide the related simulation, synthesis and formal verification tool. It is not our intention to invent yet another new HDL though we have not found an suitable language allowing us to define the RTL semantics we would like to propose. We also would like to stay language neutral to avoid inheriting something without knowing the consequence. Therefore we start by defining an new RTL language called **RTL++** that will serve the purpose as an experimental ground for studying the new semantics we proposed. The new features in the RTL++ language could help inspire designs of future HDLs or extensions of existing languages. In Section 6 we showed an example of extending SystemC to support a feature in our RTL++.

A good way to introduce a new language is by example. Listing 1 presents an RTL++ program example which we adapted from [11]. This example demonstrates most of the language constructs needed for specifying a RTL block. Similar to HDLs like Verilog module is the basic compositional unit for building structural hierarchy. A system modeled in RTL++ is composed of a list of modules that runs in parallel while communicating through their ports connected by wires. Each module is synchronous to a clock. The whole system can be seen as a globally asynchronous but locally synchronous one which suffice to represent most of the hardware systems. The detailed communication mechanism is outside the scope of this paper where we focus on the behavior of each individual *module*. The module behavior is specified with a *while true* loop statement. You can see that the language is mainly an imperative language which is familiar to most engineers. In contrast to the current practice of describing a FSM in a HDL case statement that explicitly branches to executing different actions block based on specific state values, RTL++ promotes the usage of so-called implicit state-machine style. A sequential block of program is divided into *cycle block* using a cycle delimiter - *wait* statement in RTL++. Each *cycle block* implies a distinct program control flow state.

We presents in the following sections the main ingredients in RTL++ that distinguishes it from other HDLs. Many topics of importance are omitted. These include data structures, data types, functions and procedures etc. For instance RTL++ only uses *integer* and *bool* as the only data types accepted for boolean and arithmetic operations. In this paper we focus on the execution flow control aspect of a RTL program.

3.1 Variables

To avoid the shortcomings in other HDLs such as register inferencing ambiguities, a clear semantics for synthesis is the number one objective for the design of RTL++. We define respectively three types of variables: *wire*, *register*, and *pipelined register* intended to be mapped exactly to the same type of objects in the synthesized hardware logic. They all associate with the module clock. We present their informal semantics as follows.

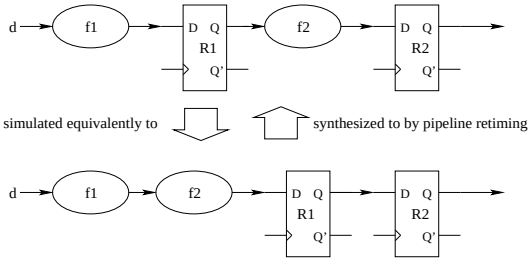

Figure 1. Using back-to-back shift registers to represent pipelined operation

3.1.1 Wire

Wire variable is used to store intermediate computation results within the *cycle block* boundary. Once assigned a new value it is immediately available to all readers. This is in a sense similar to VHDL *variable* from which it nevertheless differs in an aspect that the storage of a *wire* variable is volatile. The variable lives only in its definition *cycle block*, where there is an assignment with the variable being on the left hand side. The variable dies immediately once it passes the *cycle block* boundary delimited by *wait* statement. The *wire* variable can be mapped to the combinational output of a function in hardware logic.

3.1.2 Register

Register variable is used to store computation results across *cycle block* boundaries. It uses essentially two buffers to hold its current value and its new value. When a value is assigned to the register variable, the value is stored in the new value buffer first. Only in the beginning of next clock cycle, its current value will be updated to the saved new value buffer. Unlike VHDL signal or Verilog reg which could be synthesized to registers or wires depending on how they are inferred, a register variable in RTL++ language corresponds to a specific physical register (flip-flop) in hardware.

3.1.3 Pipelined Register

It is very awkward to capture in RTL HDL the design intent of pipelined operations which are very common in RTL design. We introduce a *pipelined register* variable in RTL++ to facilitate this purpose. As illustrated in Figure 1 a back-to-back shift register array is used to represent the effect of an pipelined operation. The depth of the array is equal to the number of stages of the pipeline. With respect to the pipeline output, it is an equivalent abstraction - the same input sequences produce the same output sequence. (This is under the normal assumption that the intermediate pipeline stage registers are not used directly for computation.) The introduction of this new type of variable in RTL++ (as the name suggests) help raise the RTL abstraction level which usually deals only with computations happening in one clock cycle. This abstraction works for modeling and simulation obviously. To be synthesizable it relies on the availability of pipeline retiming tools such as Synopsys Design Compiler [13] which has a behavioral retiming feature that can essentially find an optimal boundary in the combinational logic to insert pipeline stage registers. Also as exemplified in Listing 1 *pipelined register* variable can be used on module ports to account for interconnect delay that has become dominant for deep submicron chip design.

3.2 Abstract Syntax

We define the RTL++ language with an abstract syntax which lets us focus on the structures of the program with semantic significance rather than worrying about parsing correctly the lexical token. The syntax can be captured in a BNF-like notation as follows:

$$
\begin{array}{lr}
a ::= n \mid v \mid a_1\ aop\ a_2 & \text{(Arith. Expression)} \\
b ::= \text{true} \mid \text{false} \mid a_1\ bop_a\ a_2 & \text{(Boolean Expression)} \\
\quad \mid bop_{b1}\ b \mid b_1\ bop_{b2}\ b_2 & \\
S ::= v\ =\ a & \text{(Assignments)} \\
\quad \mid \text{wait} & \text{(Cycling)} \\
\quad \mid \text{wait until } b & \text{(Conditional Cycling)} \\
\quad \mid \text{nop} & \text{(No Operation)} \\
\quad \mid S_1;\ S_2 & \text{(Sequencing)} \\
\quad \mid \text{if } b \text{ then } S_1 \text{ else } S_2 & \text{(Branching)} \\
\quad \mid \text{while } b \text{ do } S & \text{(Loop)}
\end{array}
$$

Where the various syntactic categories and meta-variables that are used to range over constructs of each category:

n will range over numerals,
w will range over wires,
r will range over registers,
p will range over pipelined registers,
v will range over all three types of variables, $v = w \cup r \cup p$
a will range over arithmetic expressions,
b will range over boolean expressions,
S will range over statements, and
$aop \in \{+, -, *, ...\}$ a finite set of integer binary operators,
$bop_a \in \{=, <, >, ...\}$ a finite set of binary integer boolean operators,
$bop_{b1} \in \{not, ...\}$ a finite set of unary boolean operators,
$bop_{b2} \in \{and, or, ...\}$ a finite set of binary boolean operators.

$[ass]$	$\langle v = a,\ s\rangle \triangleright s[v \mapsto v.write(A[a]s)]$
$[nop]$	$\langle nop,\ s\rangle \triangleright\ s$
$[wait]$	$\langle wait,\ s\rangle \triangleright \langle (V \mapsto V.update())\rangle$
$[seq_{partial}]$	$\dfrac{\langle S1,\ s\rangle \triangleright \langle S1', s'\rangle}{\langle S1; S2, s\rangle \triangleright \langle S1'; S2, s'\rangle}$
$[seq_{complete}]$	$\dfrac{\langle S1,\ s\rangle \triangleright \langle s'\rangle}{\langle S1; S2, s\rangle \triangleright \langle S2, s'\rangle}$
$[if_t]$	$\langle$ if b then $S1$ else $S2,\ s\rangle \triangleright \langle S1, s\rangle$ if $B[b]s = \text{true}$
$[if_f]$	$\langle$if b then $S1$ *else* $S2,\ s\rangle \triangleright \langle S1, s\rangle$ if $B[b]s = \text{false}$
$[while]$	$\langle$while b do $S,\ s\rangle \triangleright$ $\langle$if b then $(S;$while b do $S)$ else nop$, s\rangle$
$[wait_{until}]$	$\langle$wait until $b,\ s\rangle \triangleright$ $\langle$while *not* b do wait$,\ s\rangle$

Table 1. Operational Semantics of RTL++

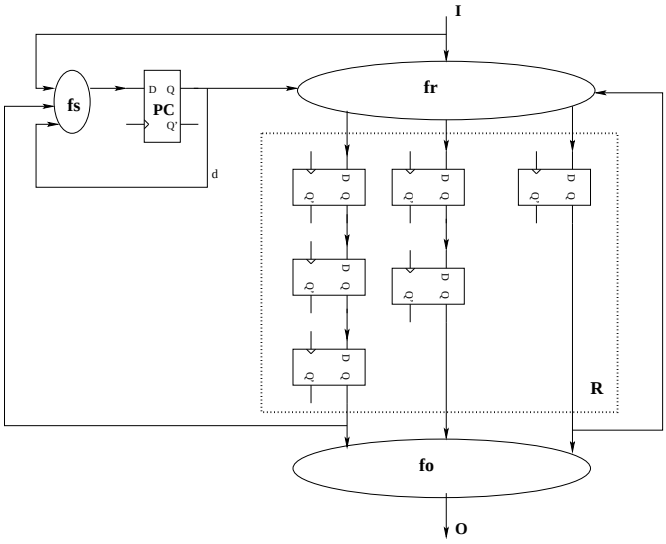

Figure 2. Register-transfer Finite State Machine

4 Formal Execution Semantics of RTL++

We define execution semantics of RTL++ using Plotkin-style structural operational semantics (SOS) as our framework. The idea is to explicitly describe how RTL++ programs compute in stepwise fashion and the possible state-transformation they perform. SOS uses a transition system which is a structure $\langle \Gamma, \triangleright \rangle$ where Γ is a set of configurations and $\triangleright \subseteq \Gamma \times \Gamma$ is the transition relation.

In the context of RTL++, the configuration is a pair $\langle S, s\rangle$ where S is the RTL++ syntactic constructs defined in Section 3.2, s is the value(state) of all RTL++ variables including wires, registers, and pipelined registers. The transition relation has two forms: $\langle S, s\rangle \triangleright \langle S', s'\rangle$ where S execution is not completed and results in an intermediate configuration S' or $\langle S, s\rangle \triangleright \langle s'\rangle$ in which S has terminated and the terminal state is s'. The specific configuration transitions for a specific RTL++ program are obtained inductively from a collection of so-called transition rules of the form $\frac{premises}{conclusion}$. If *premise* of a rule is empty it is called an axiom. The complete definition of transition relation $\triangleright$ for RTL++ is given in Table 1.

In the assignment axiom $[ass]$ we use $A[a]s$ and $B[b]s$ to denote the semantic functions for arithmetic expression and boolean expression respectively. Semantic function takes a syntactic entity as argument and returns its meaning. We will only define semantics of the variables and omit the rest of the detail of the expression semantics of RTL++. We uniformly define any of the three types of variables as a vector $\vec{\upsilon} = \{\upsilon_0, ..., \upsilon_n\}$ where for wire, $n = 0$; for regular registers, $n = 1$; for pipelined registers n is the total number of pipeline stages. We can define three variable functions as follows:

$$\begin{aligned} read() &= \upsilon_n \\ write(x) &= \{x, \upsilon_1, ..., \upsilon_n\} \\ update() &= \{undef, \upsilon_0, ..., \upsilon_{n-1}\} \end{aligned}$$

With these three functions defined, we have semantic function $A[v]s = A[v.read()]s$ which forms the basis of $A[a]s$. Axiom $[ass]$ indicates that an assignment to a variable is equal to a calling to its $write()$ function.

Axiom $[nop]$ obviously does not have any effect on the state s at all. Axiom $[wait]$ essentially expresses the update of all variables. The sequential composition rules $[seq_{partial}]$ and $[seq_{complete}]$ express that to execute $S_1; S_2$ in state s we first execute S_1 one step from s. Only when all the statements contained in S_1 has been completed S_2 can be executed. Axioms $[if_t]$ and $[if_f]$ shows that the first step in executing a branch statement is to perform the test and to select the appropriate branch. The $[while]$ rule shows that the first step in the execution of the while-construct is to unfold it one level, that is to rewrite it as a branch statement. The test will therefore be performed in the second step of the execution (where one of the axioms $[if_t]$ or $[if_f]$ is applied). The last rule $[wait_{until}]$ shows that a *wait until* statement can actually be translated into an equivalent *while* statement.

5 Synthesis Semantics: Register-transfer Finite State Machine

RTL++ has very clear synthesis semantics. All constructs of the program can be easily translated onto a hard-

ware model of computation while preserving its execution semantics. A commonly used model of computation for RTL design is FSMD [5]. However it does not support the pipelined operation semantics directly. Here we define a new model called Register-transfer Finite State Machine (RFSM) that natively supports the proposed RTL++ semantics. Figure 2 illustrates the model from a structural hardware logic point of view. Formally a RFSM is a 7-tuple

$$\langle PC, R, I, O, f_s, f_r, f_o \rangle$$

where

- $PC \subset \mathbb{Z}^*$ is a finite set of non-negative integers which essentially represent the control states (program counter) of the program. Its initial set $PC_0 = \{0\}$.
- R is the set of register (memory elements) that store the datapath states. Its initial set $R_0 = \{undef\}$.

  $$R = \{R^i | i > 0 \; and \; i \leq n\}$$

  where n is the total number of registers (pipelined registers are grouped as one), and

  $$\forall R^i \in R, R^i = (r^i[1], r^i[2], ..., r^i[m^i])$$

  where m is the number of pipeline stages of R^i, and we define the first stage register set

  $$R[1] = \{r^i[1] | i > 0 \; and \; i \leq n\}$$

  and the last stage register set

  $$R[m] = \{r^i[m^i] | i > 0 \; and \; i \leq n\}$$

- I is a finite set of input symbols.
- O is a finite set of output symbols.
- $f_s : PC \times R \times I \rightarrow PC$ is the state transition function.
- $f_r : \begin{cases} PC \times R[m] \times I \rightarrow R[1] \\ \forall R^i, \forall \; k > 1 \; and \; k \leq m^i, r^i[k] = r^i[k-1] \end{cases}$
  is the register update function.
- $f_o : PC \times R \rightarrow O$ is the output function.

Each cycle block of a RTL++ program can be labeled with a unique natural number starting with the first cycle block being 0. These cycle block labels corresponds directly into PC state register of RFSM. Synthesis tool has the freedom to re-encode the state value assignment using such as one-hot or one-cold encoding. The register and pipelined register variables also have a one-to-one mapping from their declarations in the program to the datapath register set R of RFSM. These hardware physical registers definitely can be shared among register variables having non-overlap life time [9]. This topic is out of the scope of this paper. The rest of the RTL++ assignment statements can be handled by synthesis tool using allocation and binding algorithm [5] to generate the combinational logic and the connections.

```
template < class T, const int num_stages >
class reg:sc_module {
public:
  sc_in < bool > clk;              //clock

  SC_HAS_PROCESS (reg);

  reg (sc_module_name name_)
  :sc_module (name_) {
    SC_METHOD (pipe);
    sensitive_pos << clk;
    dont_initialize ();
  };

  void write (T c) { next = c; }

  T read () { return data[num_stages - 1]; }

private:

  T data[num_stages];
  T next;

  void pipe () {
    //shifting the pipe
    for (int i = num_stages - 1; i > 0; i--) {
      data[i] = data[i - 1];
    };
    data[0] = next;
  }
};
```

Listing 2. SystemC Implementation of Pipelined Register Variable

6. Adding RTL++ Semantics in SystemC

Because formal semantics can help to clarify the ambiguities in the language design by allowing us to make rigorous statements about properties of programs and the interactions between program constructs, it provides a firm foundation for many applications related to programming languages. In this section we show an example of extending SystemC language [10] under the guidance of the new RTL++ semantics described above. More specifically we implement the key construct pipelined register variable of RTL++ as shown in Listing 2. It has the form of a new C++ class template $reg\langle T, num\_stages \rangle$.

Listing 3 provides an simple design example using this newly defined pipelined register variable type $reg\langle T, num\_stages \rangle$. This is an example adapted from the *pipe* example bundled in OSCI SystemC software package. The original example describes a 3-stage pipeline RTL design using a coding style that captures computation for each stage in a separate module and connects those modules back-to-back. We show that using the new semantics the same design intention can be kept faithfully with the output equivalence maintained on a per-cycle basis. The new code yields the cycle-to-cycle equivalent simulation result as the original OSCI example while clearly having several advantages. It has 60% less lines of code (˜110 comparing to

```
#define NUM_STAGES 3
struct alg_stages : sc_module {
    sc_in<double> in1;
    sc_in<double> in2;
    sc_out<double> powr;
    sc_in<bool>    clk;  //clock

    reg<double, NUM_STAGES> *powr_reg; //pipelined reg

    void alg();

    //Constructor
    SC_CTOR( alg_stages ) {
        powr_reg = new reg<double, NUM_STAGES>("powr_reg");
        powr_reg->clk(clk);
        SC_THREAD( alg );
        sensitive_pos << clk;
        dont_initialize();
    }
};

void alg_stages::alg()
{
    double a,b,c;
    double sum;
    double diff;
    double prod;
    double quot;

    while (true) {
        a = in1.read();
        b = in2.read();
        sum = a + b;
        diff = a - b;

        if( diff == 0 )
            diff = 5.0;

        prod = sum*diff;
        quot = sum/diff;

        c = (prod>0 && quot>0)? pow(prod, quot) : 0.;
        (*powr_reg).write(c);
        powr.write((*powr_reg).read());
        wait();
    };
}
```

Listing 3. Pipelined Register Variable Usage Example

the original ~330) and is more adaptable to design changes which happens all the time in practice. For example changing the number of stages from 3 to 4 for this example just need one change of the macro *NUM_STAGES* in our code.

7. Conclusions and Future directions

In this paper we have proposed an enhanced semantics that supports pipelined operation. Due to the lack of suitable HDL to use to define our proposed semantics, we have also defined the abstract syntax for RTL++ language intended to capture the minimal but necessary set of ingredients for RTL design modeling rather than designing a perfect language. For example, *wait* statement in RTL++ only supports one clock cycle advancement. In a real world language, to help user avoid repeating *wait* for multiple times one will certainly add another $wait[n]$ statement for coding convenience and readability improvement purpose. The execution semantics of RTL++ is documented in the intuitive and simple structural operational semantics notations. We also define a formal RFSM model for us to discuss the synthesis semantics of RTL++. These contributions form the unambiguous basis for future algorithm and tool developments in the area of RTL synthesis, formal verification and simulation. We have shown one example of extending SystemC to support the *pipelined register* variable concept in RTL++ semantics. Our immediate future research direction is to study the communication semantics between modules to investigate whether a higher level abstraction is needed for RTL.

References

[1] Accellera, http://www.eda.org/alc-cwg/cwg-open.pdf. *RTL Semantics*.

[2] Accellera, http://www.systemverilog.org. *SystemVerilog 3.1a Language Reference Manual*.

[3] G. Berry. A hardware implementation of Pure Esterel. Technical Report 06/91, Sophia-Antipolis, France, 1991.

[4] S. A. Edwards. The challenges of hardware synthesis from c-like languages. In *Proceedings of the International Workshop of Logic and Synthesis (IWLS)*, Temecula, California, June 2004.

[5] D. Gajski, N. Dutt, C. H. Wu, and Y. L. Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1994.

[6] D. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, January 2000.

[7] M. Gordon and A. Ghosh. Language independent rtl semantics. In *IEEE CS Annual Workshop on VLSI: System Level Design*, Orlando Florida, 1998.

[8] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[9] F. Kurdahi and A. Parker. Real: A program for register allocation. In *Proceedings of the 24th Design Automation Conference*, pages 210–215, 1987.

[10] Open SystemC Initiative, http://www.systemc.org. *SystemC 2.0.1 Language Reference Manual*.

[11] P. G. Paulin and J. P. Knight. Force-directed scheduling in automated data path synthesis. In *Design Automation Conference*, 1987.

[12] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[13] Synopsys Inc., http://www.synopsys.com. *Design Compiler*.

[14] S. Zhao and D. D. Gajski. Modeling a new rtl semantics in C++. In *Proceedings of ISCAS 2002*, Phoenix, Arizona, 2002.

[15] J. Zhu. MetaRTL: Raising the abstraction level of RTL design, 2001.