A Formal Model for a VHDL Subset of Synchronous Circuits

Dirk Eisenbiegler, Ramayya Kumar and Jens Müller
Forschungszentrum Informatik
(Prof. Dr.-Ing. D. Schmid)
Haid-und-Neu-Straße 10-14, 76131 Karlsruhe, Germany
e-mail: {eisen,kumar,jmueller}@fzi.de

October 23, 1995

Abstract

VHDL is based on a rather complex and powerful model, that is not very suitable for dealing with purely synchronous circuits. In this article, a synchronous subset of VHDL named $\mathcal{ABC}$-VHDL is introduced. VHDL descriptions do not always correspond to real circuit descriptions, and in general, it is difficult to figure out, whether or not a VHDL description really is an appropriate synchronous circuit description. In our approach, this can be achieved by a static analysis of the program source. In contrast to many other approaches towards synchronous VHDL subsets, the semantics of $\mathcal{ABC}$-VHDL are clear and unambiguous and furthermore conform to the standard VHDL semantics. However, the semantics are based on a far less complex timing model: RT-Level description described by output and state transition functions. $\mathcal{ABC}$-VHDL therefore is a pragmatic formal basis for the correct handling of synthesis, simulation and verification tools based on synchronous VHDL.

1 Introduction

Clear and unambiguous semantics are a basis for all tools dealing with hardware description languages such as VHDL or subsets of VHDL. Very often, semantics of hardware description languages are not as precise as they should be. Ambiguities in hardware description languages can lead to contradictory results while using different simulators or synthesis tools. Experiences have shown, that, due to the complexity of the timing model, defining exact formal semantics for VHDL as a whole is a sophisticated goal [6, 7, 8, 9].

Many tools in the area of hardware synthesis are dedicated to purely synchronous circuit descriptions. They often use VHDL subsets which are defined to describe those parts of the language that the tools can handle. Usually this is done by listing a set of restrictions and inheriting the semantics of these subsets from the VHDL standard [1]. Since VHDL is based on a very complex timing model, deriving a formal relationship between the behavioral description and a synthesized RT-level description is very difficult. In the approach presented in [4], a VHDL subset named delta-delay VHDL is defined by describing the syntax explicitly and developing the semantics from scratch. However, the semantics of delta-delay VHDL are specified in terms of a formal specification language named FOCUS, which in turn do not directly have any RT-level semantics.

In contrast to these approaches, we define a synchronous subset of VHDL called $\mathcal{ABC}$-VHDL, whose semantics are defined at the RT-level. This enables us to bridge the gap between VHDL sources and RT-level descriptions and builds a basis for the application of formal synthesis and verification methods. The paper describes the core of the $\mathcal{ABC}$-VHDL semantics, i.e. the mapping between behavioral $\mathcal{ABC}$-VHDL processes and the corresponding output- and state-transition function at the RT-level. Our approach is static in a sense that both the syntactical analysis and the mapping towards the RT-level (the semantics) is directly derived from the control structures of the process statement body. Besides behavioral descriptions, $\mathcal{ABC}$-VHDL also allows structures. The formal semantics of the structural component of $\mathcal{ABC}$-VHDL is trivial and will be omitted in this paper.

$\mathcal{ABC}$-VHDL will be formalized in a functional manner using the λ-calculus notation. We have implemented a translation from $\mathcal{ABC}$-VHDL source texts to the functional representation used in the higher order logic theorem proving environment HOL [1]. In this paper, only the main ideas of this mapping will be presented in an informal manner. For a detailed description of the formal semantics of $\mathcal{ABC}$-VHDL in

[1] HOL is a public domain theorem prover for higher order logic based on 5 axioms and 8 inference rules [5].

terms of HOL see [2]. Within HOL we have already developed synthesis specific transformations for synchronous circuits such as logic optimization, state encoding and the elimination of unreachable states based on $\mathcal{ABC}$-VHDL [3].

In section 2, the principal ideas of $\mathcal{ABC}$-VHDL and its relation to VHDL will be described. In section 3, statements will be classified in three groups named $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$, and according to this classification, we will show, how $\mathcal{ABC}$-VHDL statements and programs can be mapped to RT-level descriptions.

2 Statements and Programs in $\mathcal{ABC}$-VHDL and VHDL

In $\mathcal{ABC}$-VHDL, there is always one global clock signal. Simulation cycles are clock cycles and wait statements correspond to control states in the implementation. There are only pure input and pure output signals. Processes may either have one or zero clock inputs, and all wait statements must have the form

```
wait until clk = '1';
```

where `clk` is the clock signal of the circuit. In structural descriptions, all clock signals must be connected to the clock input of the compound circuit. The clock signal must not be connected to other signals. Simulation cycles start whenever rising slopes of the clock signal occur. During a clock cycle, the processes may read the input signals and the current variables, and depending upon these values certain variable and signal assignments are executed, and finally a new wait statement is reached. In $\mathcal{ABC}$-VHDL, only zero delay signal assignments are allowed, i.e. variable and signal assignments occur immediately when the input is read.

The transition diagram on the left hand side of figure 1 describes the life-cycle of an $\mathcal{ABC}$-VHDL process. The gray shape symbolizes a process: c_0 denotes the beginning state, $\mathsf{c}_1, \mathsf{c}_2, \ldots \mathsf{c}_n$ denote the wait statement positions and the arrows denote control state transitions. The total number of control states needed equals the number of wait statements plus one. Arrows indicate control state transitions. The behavior of processes is determined by their statement part. The control state transition selected by the process from some given control state, depends on the current state of the variables and the current input.

$\mathcal{ABC}$-VHDL processes consist of compound statements, which are recursively constructed using certain basic statements and control structures. In $\mathcal{ABC}$-VHDL, the basic statements are wait statements, signal assignments, variable assignments and the empty operation null. The control structures are: sequences of statements, loops and if-then-else structures.

Statements, i.e. parts of the statement part of processes, will also be described in a manner similar to entire processes, except that they not only have a beginning state c_0 and some wait statement positions $\mathsf{c}_1, \mathsf{c}_2, \ldots \mathsf{c}_n$ but also an end-state named c' (see figure 1).

In $\mathcal{ABC}$-VHDL, statements as well as entire programs are represented by functions that describe the transition from one clock tick to the next. These transition functions map the current control state, variable state, output state and input to the next control state, variable state and output state. The transition functions cover both the control path and the data path.

Transition functions for basic statements are rather simple. Compound statements are derived by recursively combining the transition functions of the basic statements according to the control structure of the program. Control structures are described by functions, which map such transition functions to compound transition functions. Programs are compound transition functions.

3 Formalization of Statements and Processes

VHDL processes which may reach infinite loops without ever reaching a wait statement do not correspond to real circuits. We avoid this by guaranteeing, that bodies of while statements and entire processes cannot be executed without encountering at least one wait statement. Hence we classify the statements of $\mathcal{ABC}$-VHDL into three classes named $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ and define restrictions on their combinations. In simplified terms, the differences between these three classes are as follows: On their way from position c_0 to c', the type $\mathcal{A}$ statement *never*, the type $\mathcal{B}$ statement *sometimes* and the type $\mathcal{C}$ statement *always* reaches a wait statement (see figure 2).

3.1 $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ Statements

Type $\mathcal{A}$ statements do not depend on the control state and its semantics are represented by functions that map the current variable state, output state and input to the next output state and variable state. Type $\mathcal{B}$ statements are represented by functions that map the current control state, variable state, output state and input to the next output state, control state and variable state. For type $\mathcal{B}$ statements the current state is a member of $\{\mathsf{c}_0, \mathsf{c}_1, \mathsf{c}_2, \ldots \mathsf{c}_n\}$ and the next state is a member of $\{\mathsf{c}_1, \mathsf{c}_2, \ldots \mathsf{c}_n, \mathsf{c}'\}$. Type $\mathcal{C}$ statements are similar to type $\mathcal{B}$ statements except that the function is split in two: one function describes the transition starting from c_0 and the other function

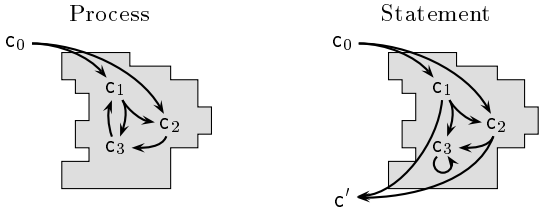

Figure 1: Control Flow in Processes and Statements

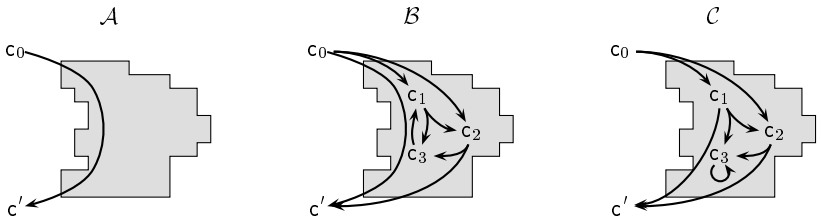

Figure 2: Classification of Statements

describes the transition starting from one of the wait statement positions $\{c_1, c_2, \ldots c_n\}$. The first function can only lead to one of the control states $\{c_1, c_2, \ldots c_n\}$ whereas the second function can also lead to c'.

3.2 Atomic Statements

There are two groups of atomic statements:

- type $\mathcal{A}$ statements: null-statements, variable assignments and signal assignments
- type $\mathcal{C}$ statements: wait statements

Type $\mathcal{B}$ statements do not correspond to any atomic statements, and arise only while combining two atomic statements into compound statements.

To describe the semantics of atomic statements, the following functional constants have been defined: null, varassign, sigassign and wait. null is a type $\mathcal{A}$ transition function that leaves both variables and output signals unchanged. Since variable assignments do not alter output signals, they can be unambiguously described by functions mapping the current input and the current variable state to the new output state. Example: The variable assignment

```
x := b;
```

in figure 4 is represented by the following function

$$\lambda\,((a, b, start), (x, y, z)).\ \ (b, y, z)$$

which maps the current input $(a, b, start)$ and the old variable state (x, y, z) to the new variable state (b, y, z), where x has been replaced by b. The function varassign maps such λ-abstracted functions to type $\mathcal{A}$ statement transition functions. Similarly signal assignments can be described by functions mapping the current input, output state and variable state to the new variable state. sigassign is used to convert them to type $\mathcal{A}$ statement transition functions.

Wait statements are type $\mathcal{C}$ statements where there is exactly one internal control state c_1. The evaluation of a wait statement starts at point c_0 then immediately reaches state c_1 and then stays there. In the next clock cycle, the process continues and immediately reaches the end-state c'. The function wait defines the type $\mathcal{C}$ transition of wait statements.

3.3 Conditions

if-then-else structures and while-loops depend on conditions. They are represented by means of functions mapping the current input and the current variable state to a boolean value. Example: The condition

```
a < b
```

in figure 4 is represented by the following function:

$$\lambda\,((a, b, start), (x, y, z)).\ \ a < b$$

This function maps the current input $(a, b, start)$ and the current variable state (x, y, z) to the boolean expression $a < b$.

3.4 Compound Statements

Based on atomic statements and conditions compound statements are derived by applying the following operators, where the letters at the end of the op-

erator names indicate which statement types are involved:

- while$\mathcal{C}$ for describing while-loops over type $\mathcal{C}$ statement bodies
- seq$\mathcal{AA}$, seq$\mathcal{AB}$, seq$\mathcal{AC}$, seq$\mathcal{BA}$,... for describing sequences of statements with arbitrary combinations of types
- ifte$\mathcal{AA}$, ifte$\mathcal{AB}$, ifte$\mathcal{AC}$, ifte$\mathcal{BA}$,... for describing if-then-else structures of statements with arbitrary combinations of types

whilec combines a condition and a type $\mathcal{C}$ statement to a type $\mathcal{B}$ statement. seq$\mathcal{AA}$, seq$\mathcal{AB}$,... are binary operators used in an infix fashion. They describe sequences for all combinations of statement types. The expression f seq$\mathcal{BC}$ g, for example, stands for the sequence of the type $\mathcal{B}$ statement f and the type $\mathcal{C}$ statement g. The result is a type $\mathcal{C}$ statement. ifte$\mathcal{AA}$, ifte$\mathcal{AB}$,... are operators mapping one condition and two statements to a compound if-then-else statement.

Figure 3 lists, how the types of compound statements are derived from the types of its parts. The term in figure 4 is constructively derived from its atomic statements and conditions by means of these operators.

In VHDL, there are also other control structures besides sequences, if-then-else statements and while loops. These control structures are nothing but syntactic sugar and can easily be constructed using these three basic control structures.

3.5 Processes

Processes can be of type $\mathcal{A}$ of type $\mathcal{C}$ only. Type $\mathcal{A}$ processes are used for describing pseudo-combinatorial circuits, i.e. combinatorial circuits, whose outputs may be buffered. Type $\mathcal{C}$ processes are used for describing sequential circuits. For type $\mathcal{A}$ processes, all inputs must be listed in the sensitivity list. Type $\mathcal{C}$ processes are sensitive to the clock signal only, and the sensitivity list must be empty. The execution of processes with type $\mathcal{C}$ statement bodies is always immediately restarted from the beginning whenever the end is reached. The operator process$\mathcal{C}$ maps a type $\mathcal{C}$ statement to a transition function of a sequential circuit and thereby removes the c' control state.

Both, type $\mathcal{A}$ and type $\mathcal{C}$ processes determine the input/output behaviour of the circuit in an unambiguous manner based on the initial state and on the output and transition function derived from the statement part.

4 Conclusions

It has been shown, that $\mathcal{ABC}$-VHDL is an appropriate synchronous VHDL subset with an unambiguous semantics that was constructed from scratch. The semantics of $\mathcal{ABC}$-VHDL is constructive in the sense that it not only specifies the relation between input and output, but also provides a mapping into RT-level descriptions. Such formal embeddings are not only a logical basis for formal argumentation (verification, formal synthesis), but are also essential for synthesis and simulation, to avoid ambiguities.

References

[1] Alain Debreil and Philippe Oddo. Synchronous designs in VHDL. In *EURO–DAC '93*, pages 486–491, Hamburg, Germany, 1993. IEEE Computer Society Press.

[2] D. Eisenbiegler, R. Kumar, and J. Müller. Formalizing the semantics for a synchronous subset of VHDL. Technical Report FZI-Report 8/95, Forschungszentrum Informatik (FZI), 1995.

[3] D. Eisenbiegler and R. Kumar. An automata theory dedicated towards formal circuit synthesis. In *Higher Order Logic Theorem Proving and Its Applications*, Aspen Grove, Utah, USA, September 1995. Springer.

[4] M. Fuchs and M. Mendler. A functional semantics for delta-delay VHDL based on focus. In *Formal Semantics for VHDL*, volume 307 of *The Kluwer international series in engineering and computer science*, chapter 1. Kluwer, Madrid, Spain, March 1995.

[5] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[6] Peter T. Breuer, Luis Sanchez Fernandez, and Carlos Delgado Kloos. Clean formal semantics for VHDL. In *EDAC '94*, pages 641–647, Paris, France, 1994. IEEE Computer Society Press.

[7] R. Boulton, A. Gordon, M. Gordon, J. Herbert, and J. van Tassel. Experiences with Embedding hardware description languages in HOL. In *Conference on Theorem Provers in Circuit Design*, IFIP Transactions A-10, pages 129–156. North-Holland, 1992.

[8] S. Olcoz and J.M. Colom. A petri net approach for the analysis of VHDL descriptions. In *CHARME93*, number 683 in Lecture Notes in Computer Science, pages 15–26, Arles,France, May 1993. Springer Verlag.

[9] W. Damm, B. Josko, and R. Schlör. A net-based semantics for VHDL. In *EURO–DAC '93*, pages 514–519, Hamburg, Germany, 1993. IEEE Computer Society Press.

sequence

predecessor	successor $\mathcal{A}$	$\mathcal{B}$	$\mathcal{C}$
$\mathcal{A}$	$\mathcal{A}$	$\mathcal{B}$	$\mathcal{C}$
$\mathcal{B}$	$\mathcal{B}$	$\mathcal{B}$	$\mathcal{C}$
$\mathcal{C}$	$\mathcal{C}$	$\mathcal{C}$	$\mathcal{C}$

if-then-else

then-branch	else-branch $\mathcal{A}$	$\mathcal{B}$	$\mathcal{C}$
$\mathcal{A}$	$\mathcal{A}$	$\mathcal{B}$	$\mathcal{B}$
$\mathcal{B}$	$\mathcal{B}$	$\mathcal{B}$	$\mathcal{B}$
$\mathcal{C}$	$\mathcal{B}$	$\mathcal{B}$	$\mathcal{C}$

while

body	while-loop
$\mathcal{C}$	$\mathcal{B}$

Figure 3: Construction Rules for Compound Statements

```
entity gcd is
  port (
    clk : in std_logic;
    a,b : in integer;
    start : in std_logic;
    ready : out std_logic;
    result : out integer
  );
end gcd;

architecture behavior of gcd is
  begin process
    variable x,y,z : integer;
  begin
    while start /= '1' loop
      wait until clk = '1';
    end loop;
    ready <= '0';
    if (a < b) then
      x := b;
      y := a;
    else
      x := a;
      y := b;
    end if;
    while (y /= 0) loop
      z := x - y;
      wait until clk = '1';
      x := y;
      y := z;
    end loop;
    ready <= '1';
    result <= x;
    wait until clk = '1';
  end process;
end behavior;
```

process$\mathcal{C}$ (
  (while$\mathcal{C}$ $(\lambda((a,b,start),(x,y,z)).\ (start/=\text{`1'}))$
    wait
) seq$\mathcal{BC}$
  sigassign$(\lambda((a,b,start),(ready,result),(x,y,z)).(\text{`0'},result))$ seq$\mathcal{AC}$
  ifte$\mathcal{AA}$ $(\lambda((a,b,start),(x,y,z)).\ a<b)$ (
    varassign$(\lambda((a,b,start),(x,y,z)).(b,y,z))$ seq$\mathcal{AA}$
    varassign$(\lambda((a,b,start),(x,y,z)).(x,a,z))$
) (
    varassign$(\lambda((a,b,start),(x,y,z)).(a,y,z))$ seq$\mathcal{AA}$
    varassign$(\lambda((a,b,start),(x,y,z)).(x,b,z))$
) seq$\mathcal{AC}$
  while$\mathcal{C}$ $(\lambda((a,b,start),(x,y,z)).\ (y=0))$ (
    varassign$(\lambda((a,b,start),(x,y,z)).(x,y,x-y))$ seq$\mathcal{AC}$
    wait seq$\mathcal{CA}$
    varassign$(\lambda((a,b,start),(x,y,z)).(y,y,z))$ seq$\mathcal{AA}$
    varassign$(\lambda((a,b,start),(x,y,z)).(x,z,z))$
) seq$\mathcal{BC}$
  sigassign$(\lambda((a,b,start),(ready,result),(x,y,z)).(\text{`1'},result))$ seq$\mathcal{AC}$
  sigassign$(\lambda((a,b,start),(ready,result),(x,y,z)).(ready,x))$ seq$\mathcal{AC}$
  wait
)

Figure 4: $\mathcal{ABC}$-VHDL Description of a GCD Circuit and its Logical Representation