

Tree Grouping for HDLA-models

Niculae Sebe

"Politechnica" University of Bucharest

Abstract

This article presents a data-structure for the representation of behavioral and structural time-continuous-systems, with an emphasis on VLSI-integrated circuits, which allows the synthesis of VHDL-A-like code.

Introduction

Simulation of heterogeneous systems on the system level implies the simulation of the whole system and of some particular devices with a special emphasis on their functional behavior.

Functional simulation can only be done efficiently by using a single system description language suitable for a single simulator. Such languages exist for the domain of digital microelectronics - VHDL or VerilogHDL. The advent of Analog HDLs like VHDL-A by Anacad offers the possibility to create behavioral models of electrical and non-electrical devices without having the limitations of Spice.

Generation of a "flat" VHDL-A-like model

In this section we will describe the possibility to generate a single VHDL-A-like model starting from two different models, as well as their interconnections. The models have to be available in the source format in order to parse their code. Simple operations as connection of two models in series and parallel (with different signal directions) are sufficient to be used recursively in order to obtain a single homogeneous VHDL-A-like model. We will call this operation *grouping*, whereby modularity is the primary advantage offered by this approach.

The accuracy of the global model reflects those of the initial ones. Despite the fact that mapping of VHDL-A-like code into the simulator-kernel or equation solver is proprietary to the CAD-vendors it can be assumed that the "flat" code will improve the speed, accuracy and stability of the performed simulation.

The grouping algorithm consists of three stages (Fig.1):

- The models are analysed lexically and syntactically, using Lex and Yacc, resulting a parse-tree.
- The models are grouped: Kirchoff's Current Law for the common pin results in a supplementary equation; Kirchoff's second law is reflected by a new state, representing the potential difference (i.e. voltage) at the common pin.
- The supplementary state and equation are eliminated.

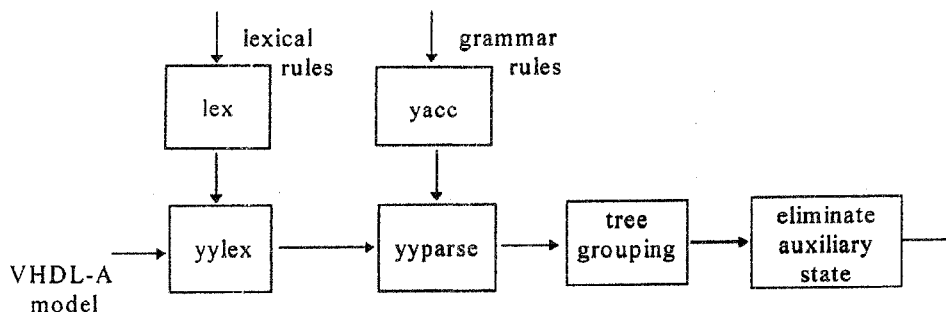


Fig.1: Diagram block

Lexical and syntactical analysis of VHDL-A-like models

Lexical and syntactical analysis of VHDL-A-like models enables an efficient and modular construction of the associated tree structure; the check for syntactical correctness of the VHDL-A-like description is another effect of this approach.

The software utilities used are Lex and Yacc. Lex is a program generator designed for lexical processing of character input streams. Lex accepts a high-level, problem-oriented specification for character string matching and produces a C routine that recognizes regular expressions. Lex divides the input into meaningful units (which are usually called *tokens*) realizing a lexical analyzer. In this case the tokens are specific words from VHDL-A-syntax as: ARCHITECTURE, ENTITY, PROCEDURAL, EQUATION, etc.

As the input is divided into tokens, one needs to establish the relationship among the tokens. This task is known as *parsing* and the list of rules that define the relationship that the program understands is a *grammar*. Yacc takes a concise description of a grammar and produces a C routine that can parse that grammar, a parser.

In our case the constructing grammar must follow the syntactic rules of VHDL-A-syntax.

Example: We consider the VHDL-A syntax having following top-view:

```
entity ::= ENTITY entity-name IS
        GENERIC (declarations)
        COUPLING (declarations)
        PIN (declarations)
        PORT (declarations)
        END [entity-name];
architecture ::= ARCHITECTURE name OF entity-name IS
                 {local-declaration}
                 BEGIN {concurrent-instruction}
                 END [name];
local-declaration ::=
    external-function | constants | variables | signals | states
concurrent-instruction ::=
    RELATION {procedure | implicit-algorithm}END RELATION;
```

The following rules describe a simplified form of the grammar:

```
entity_body : ENTITY entity-name IS
            entity_declarations
            END entity-name ';'
entity_declarations: generic_declarations
                    pin_declarations
                    ;
generic_declarations: GENERIC '(' name_list ':' analog_indications ')'
                    ;
analog_indications: ANALOG
                    ;
name_list : NAME
```

```

| name_list ',' NAME
;
pin_declarations: ...

```

The Yacc parser automatically detects whenever a sequence of input tokens matches one of these rules and performs a specific action; it also detects a syntax error whenever its input do not match any of the rules.

The main action that is performed is to build a tree structure associated with VHDL-A-model. This *parse tree* can be built by supplying actions such as:

```

expr :      expr 1 '+' expr2
      { $$ = node ( '+' , $1 , $2 ) }

```

in the specifications. That means we can call the function *node* ('+', expr1,expr2).

Composing the tree structure associated with each model

The second stage consists of composing the tree structure associated with each model, taking into account the first of Kirchoff's laws for the common pin and adding a new state that corresponds to the voltage of the eliminated pin. When eliminating this pin from the pin-list we add a supplementary equation for the new internal state. Building tree structures has the advantage that the operation, otherwise applied to VHDL-A-text, is done using a tree (it is easier to add and delete a node, to modify node parameters, etc.).

Example: Suppose we have to link two RC-circuits (Fig.2):

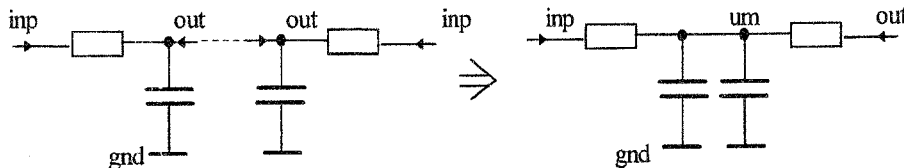


Fig.2: Grouping RCs

The initial model is described by the following VHDL-A-like description:

```

ENTITY RC IS
  GENERIC (R,C : analog);
  PIN (inp, out, gnd : electrical );
END ;

ARCHITECTURE a OF RC IS
  STATE ic: analog ;
  BEGIN
    RELATION
      PROCEDURAL FOR init =>
        ic:= 0.0 ;

```

```

PROCEDURAL FOR dc, transient =>
    [out,gnd].i %= ic-[inp, out].v/ R;
    EQUATION ( ic ) FOR transient =>
        C * [out , gnd].v == integ( ic );
    END RELATION ;
END;

```

After this stage we obtain the following VHDL-A-text:

```

ENTITY RC2 IS
    GENERIC (R,C : analog);
    PIN (inp1, out2, gnd : electrical );
END ;

ARCHITECTURE a OF RC2 IS
    STATE ic1,ic2,um: analog ;
    BEGIN
        RELATION
            PROCEDURAL FOR init =>
                ic1:= 0.0 ; ic2 :=0.0 ;
            EQUATION ( ic1,ic2,um ) FOR transient =>
                ic1- (inp.v -um )/R + ic2 -(out.v - um )/R == 0.0;
                C * um == integ( ic1 );
                C * um == integ( ic2 );
            END RELATION ;
        END;

```

A supplementary state *um* has been introduced and a new corresponding equation.

Elimination of the auxiliary internal state and the corresponding equation

This stage is imposed by obvious complexity reasons.

We use Mathematica's facilities for symbolic calculus. We have to solve symbolically a system of equations to eliminate the state and the corresponding equation. Therefore we use MathLink to make some literal operations in Mathematica and to receive the result.

Conclusion

We have developed a simple method to obtain a tree structure for a global model consisting of the two models that are grouped. The obtained tree structure can be used to generate the text of the resulting VHDL-A model and also to build a structure to be used as a pre-processing part in VHDL-A model compilation. The gain of this approach is that the model-generation is decoupled from writing and learning VHDL-A-syntax, therefore a graphical model entry would be a straight-forward continuation of this work.

Bibliography

- J.Levine, T.Mason, D.Brown: "lex & yacc", O'Reilly&Associates, 1992
- HDL-A Language Reference Manual , ANACAD ,March 1994
- S.Wolfram: Mathematica , Addison Wesley 1992