A Case-Based Reasoning Approach for the Automatic Generation of VHDL-AMS Models

Ahmad Al-Kashef
Mentor Graphics
ahmad_al-kashef@mentor.com

Manal M. Zaky
Ain Shams University
manalmourad@yahoo.com

Mohamed Dessouky
Mentor Graphics
mohamed_dessouky@mentor.com

Hassan El-Ghitani
Misr International University
hassan.elghitani@miuegypt.edu.eg

ABSTRACT

Automatic generation of analog and mixed-signal (AMS) behavioral models from specifications is an important component of top-down design methodologies. In this paper, we present an expert system solution to this challenge. Based on a representation of the model functionality, our expert system manipulates a library of previously developed models to synthesize a new model. Automatically generated VHDL-AMS behavioral models are shown to be of expert quality.

1. INTRODUCTION

The adoption of top-down design methodologies fueled the need for the automatic generation of behavioral models which started as (and still mostly is) a manual development process.

Roychowdhury [1] lists several broad methodologies for automated behavioral modeling including algorithmic methods, symbolic methods, black-box methods (such as data-mining, neural networks, genetic algorithms and multi-dimensional tables) and automation of the manual modeling process. All of these methods, except for the automation of the manual modeling process, are bottom-up methods (i.e. they start from a given circuit description, often a netlist, to develop a behavioral model). A survey of these methods is given in [2]. In top-down design methodologies though, model creation precedes circuit design and circuit specifications are a product of design space exploration which requires behavioral models.

Attempts to automate the manual modeling process starting from model specifications (rather than a given netlist) are relatively fewer compared to bottom-up approaches. Authors in [3] automated a manual modeling process for continuous-time Delta-Sigma modulators. An analog behavioral model synthesizer described in [4] expresses model behavior in the form of functional diagrams drawn as an interconnection of symbols. Each symbol stands for an elementary analog behavior. Behavioral elements are coded separately and then combined to generate the model HDL-A code. Two approaches for the generation of behavioral VHDL models from descriptions written in natural language are presented in [5].

The most prevalent approach towards creating behavioral models in industry though, is manual abstraction [1]. Manual efforts start with an understanding of the circuit but do not require the circuit netlist to develop an equivalent model. Manual abstraction (e.g. [6] - [9]) usually results in simulation speedups orders of magnitude higher than automated efforts (e.g. [10] - [12]). Although fast-SPICE simulators don't provide parameterized models, they achieve speedups similar to those achieved by automated modeling.

The development of parameterized, AMS behavioral models for all circuit classes including strong nonlinear behaviors, with multiple-inputs and multiple-outputs, which achieve high speed gains and do not require the circuit netlist as an input, is a manual process reserved to modeling experts working side-by-side with design teams. An attempt to automate this development process naturally lends itself to an expert system solution due to the lack of an algorithmic solution which meets all these requirements.

Case-based reasoning (CBR) is an approach to knowledge-based problem-solving employed in expert systems [13] that has been commonly used in code understanding and generation. A survey of CBR techniques for CAD systems used in design is given in [14], and a review of the applicability of these techniques to code generation is given in [15]. A CBR system which generates digital models is given in [16].

In this paper, we present an architecture for a case-based reasoner which automates the generation of VHDL-AMS behavioral models for AMS circuits (Section 2). Section 3 describes the knowledge representation on which our work is based. Due to the limited space of the paper, we will focus on how the CBR adapts VHDL-AMS models (Section 4). Two examples of generated models are then listed and discussed in Section 5. The whole system is fully described in [17].

2. CBR ARCHITECTURE

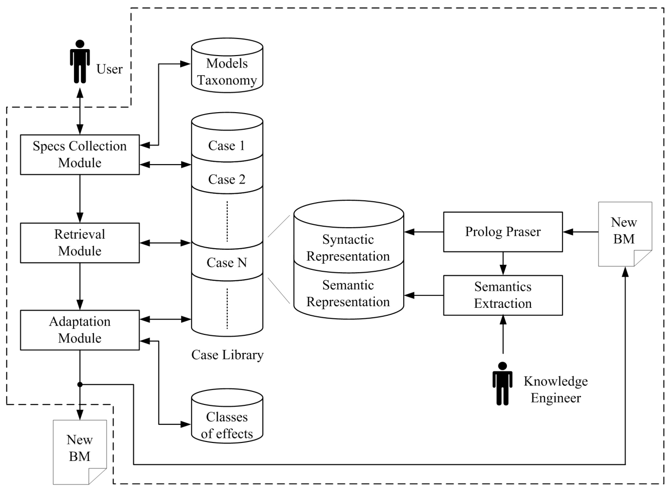

Figure 1: CBR architecture developed for model generation.

Figure 1 illustrates the architecture of the developed CBR system. The Case Library is a database of VHDL-AMS models' representations. Each case represents a single model. Cases are indexed according to their salient features (interface and behavior). Using knowledge about different circuit architectures and levels of abstraction stored in the Models Taxonomy database, the Specs Collection Module collects specifications for a behavioral model through a graphical user interface (GUI). The user specifies the model's class (VCO, DAC, ADC, etc.), ports and their functions (input, output, reference, ground, etc.), and model behavior along with the associated generics. Specifications are then passed to the Retrieval Module which relies on an adaptation-guided retrieval algorithm [13]. The algorithm is based on heuristic rules expressed in Prolog predicates and is biased to select cases most easily adaptable to the current situation. The retrieved case is compared against the given specifications to determine the adaptation effort required, and the Adaptation Module manipulates the retrieved case to synthesize the specified model. The new model is finally fed back into the CBR to be added to the Case Library.

The adaptation of VHDL-AMS models is accomplished by manipulating a Prolog representation of the VHDL-AMS source obtained by an open source SWI-Prolog Parser [18] with the help of a set of Prolog rules in the Adaptation module and the required knowledge for the specified adaptation as will be explained in detail in the next section. The modified Prolog representation is finally transformed back to readable VHDL-AMS source code.

The output of the Prolog Parser also serves as an input to the Semantics Extraction module which helps the knowledge engineer augment the syntactic representation with semantics about the model's functionality.

We used XPCE, SWI-Prolog's native GUI library [19] to implement the GUI, and SWI-Prolog ODBC interface [20] in the Retrieval module to issue SQL queries to an Access database.

3. KNOWLEDGE REPRESENTATION

In case-based reasoning, knowledge representation covers the cases stored in the expert system's database (case representation), and the adaptation strategies used to adapt these cases to fit a new situation. Each case has a semantic and a syntactic representation. The syntactic representation is the Prolog parse tree substitute of the VHDL-AMS source code which enables its manipulation. The rest of this section deals with the semantic representation.

Semantic representation captures the qualitative knowledge about a behavioral model. Our representation is based on the idea that total circuit behavior can be decomposed into a number of separable behaviors. Each of these behaviors is called an effect. Effects are classified into ideal, non-ideal, and auxiliary, only in relation to a given model. An ideal effect represents the ideal behavior of the circuit (e.g. amplification in a amplifier). A non-ideal effect represents a particular non-ideality in the behavior (e.g. propagation delay in an AND gate) and an auxiliary effect complements an ideal effect inside the model, but is not in itself an ideal effect (e.g. binary-to-decimal conversion in digital-to-analog converters). Besides effects, an AMS behavioral model is also represented in terms of some macro-modeling elements, and the heuristic rules that experts use to glue these effects and macro-models together inside the architecture of a model.

This representation is captured into a graphical conceptual model [13] which is translated into Prolog predicates and augmented with additional semantics about the model's effects, generics, and ports. An example diagram of a generated digital-to-analog converter (DAC) model (Section 5) is shown in Figure 2.

The border of the conceptual diagram represents the model entity, while the model's behavior (architecture) is captured inside the border of the diagram. Behavior is decomposed into effects (rectangular boxes) and macro-modeling elements (vout and iout). Unipolar digital-to-analog conversion is the ideal effect. Delay and DC shift are non-ideal effects, while binary-to-decimal conversion and extraction of converter resolution are auxiliary effects.

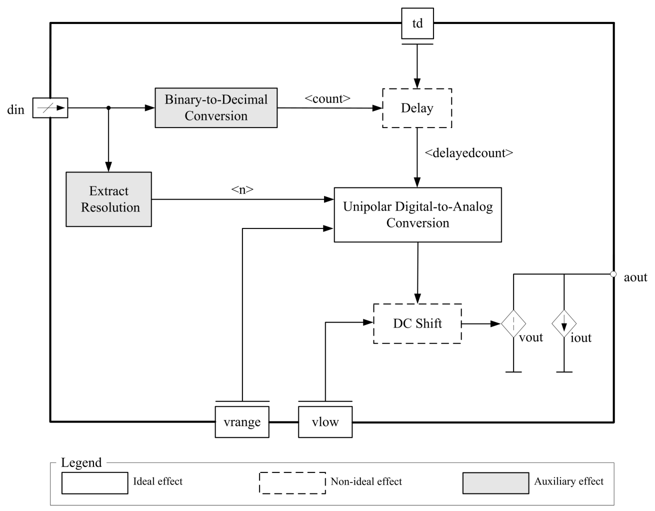

Figure 2: Conceptual diagram of a DAC behavioral model.

Arrows represent VHDL-AMS objects such as signals, terminals, and quantities declared inside the model architecture.

Classes of effects are stored in a separate library and instances of these classes are called in each of the cases inside the CBR case library. Each effect class consists of a VHDL-AMS source code template, VHDL-AMS objects used inside the template, and effect-specific adaptation rules to enable the integration of this effect inside any behavioral model. An example of a delay effect class is given in Listing 1.

Listing 1: Delay effect class.

```
%effect_class(ClassName,GenericList,Inputs,Outputs,IntermediateVars
,
% PrologCode,RulesForAddingEffect,RulesForRemovingEffect).
effect_class(delay,[Td],[In1],[Out1],[],code,AddRules,RemoveRules):-
code=vhdl_process(_,_,null,[In1],[],
[vhdl_if(null, null, rel(=, vhdl_call(domain, []),
vhdl_call(quiescent_domain, [])),
[signal(null, Out1, null, null, [event(vhdl_call(In1, []), null)])],
[signal(null, Out1, null, null, [event(vhdl_call(In1, []),
vhdl_call(Td, []))])])]),
AddRules=[concat_atom([In1,'_',delay,'_',time],Td),
concat_atom([delayed,'_',In1],Out1),
generics_types([Td],[real]),
var_type([Out1],[type(In1)]),
rename(In1,Out1)],
RemoveRules=[rename_variable(Out1,In1)].
```

4. ADAPTATION

We implemented four basic adaptation strategies: Removing an effect, adding an effect, output current to output voltage transformation and vice versa, and single-ended operation to differential operation transformation and vice versa.

The conceptual diagram represented in Figure 2 intuitively exposes the basic procedures used to implement these strategies.

4.1 Remove an Effect

The simplified pseudo-code of the algorithm used to remove an effect from a given model is:

1- Remove the code template corresponding to the effect to be removed from the model.

2- Remove associated generics if they are not also associated with other effects.

3- Depending on the adaptation rules of the effects preceding and following the effect to be removed, replace the output object of the effect to be removed with its input object, or vice versa, and remove the declaration of the substituted object.

4.2 Add an Effect

To add an effect to a given model, the following steps are executed:

1- Retrieve the effect to be added from the effects' classes library.

2- Determine an insertion point by consulting the user about the right sequence of effects.

3- Add generics associated with the new effect class to the model's generic list.

4- Declare the output object of the effect to be added.

5- Add the effect code template to the model's architecture.

6- In the added effect code template, replace the input object with the output object of the effect preceding the one added.

7- In the code template of the effect following the added one, replace the input object with the object declared in step 4.

4.3 Output Current to Output Voltage Transformation and Vice Versa

Figure 3 illustrates the idea behind the current-to-voltage transformation. The opposite is similar. AMS behavioral models usually have an output stage whose macro-model representation is that of a current (voltage) source.

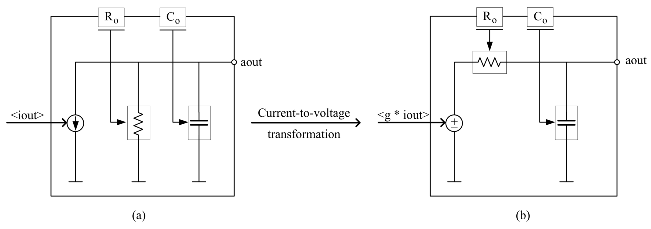

Figure 3: Conceptual diagram of a current source output stage (a) and its equivalent voltage source output stage (b).

This stage can be thought of as an effect whose inputs are the values of the current (voltage) source and the output resistance and capacitance, and whose output is the output terminal. The values of the current (voltage) source of the output stage would be the result of some computations performed in the model according to its functionality; the output of a previous effect.

This adaptation strategy substitutes a current (voltage) output stage effect with an equivalent voltage (current) output stage effect, and multiplies the input of the effect by a gain quantity which is added to the model's list of generics.

4.4 Single-ended Operation to Differential Transformation and Vice Versa

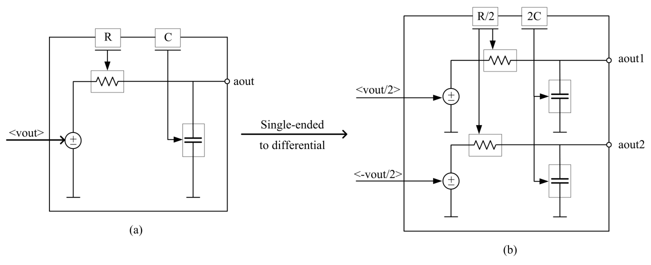

Figure 4: Single-ended (a) to differential (b) transformation.

This adaptation strategy builds on the notion explained in the above section; an output stage of an AMS behavioral model can be treated as an effect. Figure 4 illustrates how a single-ended output stage is transformed into a differential output stage. The opposite is similar. To make the transformation from single-ended to differential operation and vice versa the adaptation module substitutes stages the same way it substitutes effects.

The retrieval and the adaptation algorithms enable the generation of models whose types are not represented in the case library. For example, the case library may contain an AND gate model with a propagation delay non-ideality, but lacks a delay element model. Upon the user's request to generate a delay element model, the CBR is capable of reusing the non-ideal delay effect implemented in the AND gate inside the delay element (as an ideal effect).

5. EXAMPLE

An automatically generated DAC model is given in Listing

Listing 2: Automatically generated DAC model.

```
entity d2a is
  generic (td        : time := 1.0 ns;
          -- trise    : real := 0.0e-9;
          -- tfall    : real := 0.0e-9;
             vrange : real := 5.0;
             vlow     : real := 0.0);
  port (terminal aout : electrical;
          signal     din   : in  std_logic_vector);
  constant n : integer := din'length; -- Extract Converter Resolution
end d2a;
architecture machine of d2a is
  quantity vout across iout through aout;
  quantity vdac              : real := 0.0;
  signal     delayedcount : real := 0.0;
  signal     count            : real := 0.0;
-- quantity rampedcount : real := 0.0;
begin
  vout == vdac + vlow; -- DC Shift
  vdac == vrange / 2.0 ** n * delayedcount; -- D2A Conversion
  -- vdac == vrange / 2.0 ** n  * rampedcount;
  break on delayedcount;
  -- break on rampedcount;
  -- rampedcount == delayedcount'ramp(trise,tfall);
  binary2decimal : process (din) is
    variable countvar :  real := 0.0;
  begin
    countvar := 0.0;
    if din'ascending then
      for i in din'range loop
        if din(i) = '1' then
          countvar := countvar + 2.0 ** (din'high - i);
        end if;
      end loop;
    else
      for i in din'range loop
        if din(i) = '1' then
          countvar := countvar + 2.0 ** (i - din'low);
        end if;
      end loop;
    end if;
    count <= countvar;
  end process binary2decimal;
  delay : process (count) is
  begin
    if domain = quiescent_domain then
      delayedcount <= count;
    else
      delayedcount <= count after td;
    end if;
  end process delay;
end machine;
```

2 as an example of the first adaptation strategy; the adaptation module is able to remove the ramping effect on the converter’s output (lines number 3, 4, 16, and 23 are removed, line 19 substitutes line 20, and line 21 substitutes line 22). This effect is part of the representation of the model selected from the case library.

An example of the second adaptation strategy in given in Listing 3, where a delay effect was added to a switched current cell (lines number 2, 8, 11-18 are added, and line 19 substitutes line 20).

Listing 3: Automatically generated switched current cell.

```
01 entity currentCell is
02   generic (ctrl_delay_time : time := 10 ns;
03            iout            : real  := 0.001);
04   port (signal   ctrl     : in std_logic;
05         terminal aout : electrical);
06 end currentCell;
07 architecture machine of currentCell is
08   signal delayed_ctrl : std_logic;
09   quantity v_out across i_out through aout;
10 begin
11   process (ctrl) is
12   begin
13     if domain = quiescent_domain then
14       delayed_ctrl <= ctrl;
15     else
16       delayed_ctrl <= ctrl after ctrl_delay_time;
17     end if;
18   end process;
19   if delayed_ctrl = '1' use
20   -- if ctrl = '1' use
21     i_out == iout;
22   else
23     i_out == 0.0;
24   end use;
25   break on delayed_ctrl;
26 end machine;
```

To evaluate the output of our expert system, we compared its results to those obtained by a human expert in AMS behavioral modeling.

The main difference between the automatically generated code and that of a human expert is the separability of effects. Each VHDL-AMS statement generated by the CBR contribute to one, and only one effect, and then effects are glued together by free quantities and signals. A human expert on the other hand tends to mix different effects into a single statement and thus uses a fewer number of quantities and signals in the model’s architecture. A portion of the DAC model developed by a human expert is given in Listing 4 as an example (equivalent to lines number 18, 19, and 43-50 in Listing 2). According to ADVance MS^{TM} user manual [21], using intermediate quantities does not degrade the simulation performance.

Listing 4: Portion of the DAC model developed by a human expert.

```
Vout == Vrange/2.0**N*count + Vlow;
```

A quantitative comparison of simulation performance was carried out using simulation statistics provided by ADVance MS in a typical test case. Comparison results of the DAC models are illustrated in Table 1 and those of the switched current cell are illustrated in Table 2. The purpose of this comparison is to insure that automatically generated models are as efficient as those developed by human experts.

Table 1: Simulation performance comparison of the DAC models.

	Generated Model	Human Expert Model
Total CPU time	40 ms	50 ms
Memory used (in KB)	49,772	49,772
Number of digital kernel events	54	60
Accepted analog time steps	340	340
Rejected analog time steps	8	8

Table 2: Simulation performance comparison of the switched current cell models.

	Generated Model	Human Expert Model
Total CPU time	40 ms	50 ms
Memory used (in KB)	49,756	49,756
Number of digital kernel events	33	33
Accepted analog time steps	351	225
Rejected analog time steps	9	0

The electrical correctness and accuracy of the generated models were insured by comparing the output waveforms of the generated models against those developed by a human expert. Waveforms perfectly coincide.

6. CONCLUSION

The presented expert system is capable of generating parameterized, linear and non-linear VHDL-AMS behavioral models for any class of circuits. A couple of automatically generated behavioral models were given as an example. Qualitative and quantitative assessments of these models present an evidence of their expert-quality. A bottleneck in the usage model of any CBR system is the availability of cases. Our CBR makes use of the hundreds of VHDL-AMS models available with ADVance MS. These models are laden with a very rich set of AMS effects.

Our CBR cannot generate hierarchal models (models built up of another behavioral models). Our future work tackles this limitation. We are also working on extending the set of adaptation strategies and developing a graphical editor which enables the user to directly manipulate the conceptual diagram representation.

REFERENCES

[1] Jaijeet Roychowdhury, “Automated macromodel generation for electronic systems,” in *Proceedings of the 2003 International Workshop on Behavioral Modeling and Simulation,* 2003, pp. 11-16.

[2] H. A. Mantooth, L. Ren, X. Huang, Y. Feng, and W. Zheng, “A survey of bottom-up behavioral modeling methods for analog circuits,” in *Proceedings of the 2003 International Symposium on Circuits and Systems,* 2003, pp. III-910- III-913 vol.3.

[3] K. Francken, M. Vogels, E. Martens, G. Gielen, “A behavioral simulation tool for continuous-time delta sigma modulators,” in *IEEE/ACM International Conference on Computer Aided Design,* 2002, pp. 234- 239.

[4] V. Moser, H.-P. Amann, and F. Pellandini, “Behavioural modelling of analogue Systems with ABSynth,” in *Analog and Mixed-Signal Hardware Description Languages*, A. Vachoux, J.-M. Bergé, O. Levia, and J. Rouillard, Kluwer Academic Publishers, 1997, pp. 93-100.

[5] W. R. Cyre, J. Armstrong, M. Manek-Honcharik, and A. J. Honcharik, “Generating VHDL models from natural language descriptions,” in *Proceedings of the Conference on European Design Automation*, 1994, pp. 474-479.

[6] D. El-Ebiary, Maged Fikry, Mohamed Dessouky, and Hassan Ghitani, “Average behavioral modeling technique for switched-capacitor voltage converters,” in *Proceedings of the 2006 IEEE International Behavioral Modeling and Simulation Workshop,* 2006, pp. 109-114.

[7] D. El-Ebiary, Mohamed Dessouky, and Hassan Ghitani, “Behavioral modeling of a charge pump voltage converter for SoC functional verification purposes,” in *Proceedings of the 2007 IEEE International Behavioral Modeling and Simulation Workshop,* 2007, pp. 84-89.

[8] N. H. Saada, R. S. Guindi, and A. E. Salama, “A new approach for modeling the nonlinearity of analog to digital converters based on spectral components,” in *Proceedings of the 2006 IEEE International Behavioral Modeling and Simulation Workshop,* 2006, pp. 120-125.

[9] Mentor Graphics, Appl. Note 10201.

[10] L. Nathke, V. Burkhay, L. Hedrich, and E. Barke, “Hierarchical automatic behavioral model generation of nonlinear analog circuits based on nonlinear symbolic techniques,” in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, 2004, pp. 442- 447 Vol.1.

[11] Yasunori Miyahara, John Moore, Taichi Ikedo and Lars Andersen, “Automatic behavioral model generation suite for mobile phone system analysis,” Agilent Technical Publications, 2003.

[12] Carsten Borchers, Lars Hedrich, and Erich Barke, "Equation-based behavioral model generation for nonlinear analog circuits," in *33rd Annual Conference on Design Automation*, 1996, pp. 237-240.

[13] George F Luger and William A Stubblefield, *Artificial Intelligence: Structures and strategies for complex problem solving*, 3rd ed. Boston: Addison Wesley Longman, 1998.

[14] B. Trousse, and W. Visser, “Use of case-based reasoning techniques for intelligent computer-aided-design systems,” in *Proceedings of the International Conference on Systems, Man, and Cybernetics*, 1993, pp. 513-518 vol.3.

[15] Broad A. and Filer N., “Applying case-based reasoning to code understanding and generation,” in *Proceedings of the Fourth United Kingdom Case-Based Reasoning Workshop*, 1999, pp. 35-48.

[16] P. Gomes and C. Bento, “A case similarity metric for software reuse and design,” in *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 2001, pp. 21-35.

[17] Ahmad Al-Kashef, “A case-based reasoning approach for the knowledge representation of VHDL-AMS behavioral models,” M.S. thesis, Ain Shams University, Cairo, Egypt, 2008.

[18] VHDL-1076.1 Parser/Pretty-Printer in SWI-Prolog, http://www.cs.wright.edu/~tkprasad/VHDL-AMS/README.html.

[19] XPCE: the SWI-Prolog native GUI library, http://www.swi-prolog.org/packages/xpce/.

[20] SWI-Prolog ODBC Interface, http://www.swi-prolog.org/packages/odbc.html.

[21] *ADVance MS user Manual*, Mentor Graphics, 2008.