IBMG: Interpretable Behavioral Model Generator for Nonlinear Analog Circuits via Canonical Form Functions and Genetic Programming

Trent McConaghy
ESAT-MICAS
K.U. Leuven
Leuven, Belgium

Georges Gielen
ESAT-MICAS
K.U. Leuven
Leuven, Belgium

***Abstract*—This paper presents IBMG, an approach to generate behavioral models of nonlinear analog circuits, with the special distinction that it generates models that are compact, *interpretable* expressions, which do are not restricted to any pre-defined functional templates. IBMG outputs a small set of interpretable nonlinear differential equations that approximate the time-domain behavior of the circuit being modeled. The approach uses genetic programming (GP), which evolves functions, but GP has been heavily modified so that the behavioral expressions follow a special "canonical functional form" grammar to remain interpretable. IBMG has explicit error control: it provides a *set* of models that trade off complexity and accuracy. Experimental results on a strongly nonlinear latch circuit demonstrate the usefulness of IBMG.**

I. Introduction

Fast, effective system-level analog design practices are becoming increasingly important due to the rise in the use of mixed-signal ICs and communication circuits in particular. How sub-circuits are modeled for design and verification heavily influences speed and risk of system-level design. Using SPICE to simulate a single system-level design can easily take hours or days, which makes verification painful and automated sizing infeasible.

A potential means to speed up simulation is via behavioral models which approximate the dynamic behavior of sub-circuits but simulate orders of magnitude faster [1]. These models can be manually created. This makes them understandable (because they are interpretable expressions, and a human generated them), and therefore somewhat trustworthy. However, creating of such models takes weeks to years, and even then the validity of the model could expire with new technology generations.

An alternative is to generate models automatically. Model order reduction (MOR) and regression are the two main approaches. MOR uses projections to transform the system's states into a smaller set that retain the essence of the system's behavior. Early MOR research focused on linear circuits, but more recent research has tackled weakly nonlinear circuits and finally strongly nonlinear circuits. [2] is a recent survey. For nonlinear circuits, piecewise linear [3] and piecewise polynomial approaches [4] each tie together a group of linear or polynomial models along likely trajectories in state space. Kernel methods [5][6] transform the nonlinear state space into a high-dimensional linear feature space which is more readily handled. MOR uses the circuit's internal dynamics to retain some trust; however, since they are not interpretable they can be less preferable to many designers. Also, they are subject to the biases imposed by the particular choice of regression. Specifically, polynomials extrapolate poorly; distance-based kernels (e.g. radial basis kernels) and inner product kernels place equal importance on all state variables even though some state variables may affect the dynamics far more than others. That is why [5] mentioned high sensitivity of model error to the scaling of the state variables. Finally, nonlinear MOR approaches divide the problem into two sub-problems that are solved sequentially (finding a projector, finding the corresponding regressor); the only way this division could be optimal is if the two sub-problems are completely independent, and there is no reason to believe this is so. Thus, the approaches sacrifice their chances of optimality, most notably the compressibility aspect (which directly influences interpretability).

On the other hand, regression or "black-box" approaches create models that learn from the simulation waveforms of the circuit's inputs and outputs [7]. The topology and internal states of the circuit are not considered. As the problem is not sub-divided into sub-problems, it has (at least the theoretical) opportunity to optimally capture the dynamics. The models can be compact. However, they are not interpretable, which impedes designer trust. Because they do not aim to project the dynamics of the detailed system onto a smaller system, they do not "gain back" some trust.

This paper presents IBMG: Interpretable Behavioral Model Generator. IBMG uses genetic programming [8], which evolves functions, but modified so that differential equations follow a special canonical form. Trust is gained because the models are readily-interpretable equations. IBMG creates compositions of functions that best fit the problem at hand; i.e. is not constrained to any specific basis function nor any predefined functional template.

This paper is organized as follows. Section II describes the problem; section III provides background on GP; sections IV and V describe IBMG and its grammar. Section VI provides experimental results, followed by the conclusion.

II. Problem Description

We consider a p-input q-output nonlinear dynamical system, specifically a circuit, of the form:

$$\frac{dx}{dt} = f(x(t), u(t)) \qquad (1.1)$$
$$y(t) = Cx(t) + Du(t)$$

where *x(t)* is the system's *n*-dimensional state (i.e. node voltages and branch currents in the circuit), *u*(*t*) is the *p* inputs at time *t*, and *y*(*t*) is the *q* outputs at time *t*. *f*(*x*, *u*) is an arbitrary nonlinear function vector field that describes how the state changes. *y*(*t*) is a linear function of *x*(*t*) and *u*(*t*).

The task of the behavioral modeling system is to create a more compact form of the given dynamical system, i.e. one with *m* states where $m \ll n$. The model must be interpretable *behavioral expressions*, i.e. easily readable functional forms that describe how the state changes. Finally, the approach must have error control by actually generating a *set* of models that trade off between error and complexity. The generator's inputs are *u*(*t*) and *y*(*t*), taken from a transient simulation using a standard SPICE simulator. With the aim of interpretability, *x*(*t*) is *not* an input, even though it creates a more difficult learning problem. The expressions to be generated must take the form:

$$\frac{dz}{dt} = g(z(t), u(t)) \qquad (1.2)$$
$$y(t) = Ez(t) + Fu(t)$$

where *z* is the system's state, and *g*, *E*, and *F* are the reduced-system equivalents of f, *C*, and *D* respectively. Initial system state is set to be *z*(0) = (0,0,…). IBMG must "learn" the vector valued function *g*(*z*, *u*) as well as *E* and *F*. Learning E and F is merely a set of linear learning problems (one for each output variable) once *z*(t) for each t is known. Learning *g*(*z*, *u*) is the major challenge, as each point g in the search space of possible *G's* involves a choice of the number of basis functions, and the functional form of each of those basis functions (which takes the other basis functions and u as an input). Any possible composition of functions is allowed.

We could have formulated the problem more generally, i.e. *y* as a *nonlinear* function of *x* and *u*. But IBMG approximates nonlinear mappings via state variables that do not appear in *f*(), which relate *x* and *u* to *y* in a nonlinear fashion. In making this choice we simplify IBMG and also encourage re-use of expressions for outputs.

Let us define our goals more specifically. We want to generate a set of models that trade off according to these objectives:

$$\begin{cases} \min\ nmse_{tot}(y, y_h) \\ \min\ complexity(h) \end{cases} \qquad (1.3)$$
$$h \in H$$

where *h* is a model in model space *H*; a given *h* is composed of a *g*, *E*, and *F*. $nmse_{tot}$ is defined as follows:

$$nmse_{tot}(y, y_g, dy, dy_g) = \frac{1}{p}\sum_{i=1}^{p}\frac{1}{2}\left(nmse(y, y_g) + nmse(dy, dy_g)\right) \quad (1.4)$$

where *dy*(t) = y(*t*)-y(*t-dt*) and *nmse* follows the usual definition:

$$nmse(y_{ref}, y_2) = \frac{1}{len(y_{ref})}\sqrt{\sum_{j=1}^{len(y_{ref})}\left(\frac{y_{ref}(j) - y_2(j)}{\max(y_{ref}) - \min(y_{ref})}\right)^2} \quad (1.5)$$

where *len*(*y*) is the number of samples in *y*.

The speed of model building is *not* considered a goal at this point.

III. Background: Genetic Programming

Genetic Programming (GP) [8] is an evolutionary algorithm, with the distinguishing characteristic that GP individuals (points in the design space) are *trees*. While GP has been used previously to evolve differential equations [9], this paper is the first to have an up-front emphasis on interpretability, to create tradeoffs of complexity and error, and to model circuits. The functional form of results from canonical GP is completely unrestricted, hindering interpretability. This is exacerbated by the observation that GP-evolved functions tend to bloat with improved fitness [10]. The main challenge is to find a way to restrict the form enough to make the expressions interpretable, without actually constraining away from any possible functional forms.

IV. Description Of IBMG

IBMG uses an altered form of CAFFEINE [11]. CAFFEINE took GP as a starting point, but extended it to properly address creating static mappings that were interpretable. It attacked interpretability via a multi-objective approach that provides a tradeoff between error and complexity, and most notably a specially designed grammar to constrain search to specific functional forms. IBMG uses a state-of-the-art *multi-objective* evolutionary algorithm, NSGA-II [12], to generate a set of models that, collectively, trade off $nmse_{tot}$ and complexity. "Complexity" is dependent on the number of basis functions, the number of nodes in each tree, and the exponents of "variable combos" (VCs; more on that later). It is measured as:

$$\text{complexity}(f) = \sum_{j=1}^{M_f}\left(w_b + \text{nnodes}(j) + \sum_{k=1}^{\text{nvc}(j)}\text{vccost}(vc_{k,j})\right) \quad (6)$$

where w_b is a constant to give a minimum cost to each basis function, nnodes(*j*) is the number of tree nodes of basis function *j*, nvc(*j*) is the number of VCs of basis function *j*, and $\text{vccost}(\text{vc}) = w_{vc}\sum_{\text{dim}=1}^{d}\text{abs}(\text{vc}(\text{dim}))$. IBMG does *simplification during generation* via pressure to lower complexity. The user avoids *a priori* decisions on the error and complexity because IBMG provides a tradeoff set of alternatives.

V. Grammar and Operators

A grammar can constrain the GP search space, as in [13]. Evolutionary operators respect the derivation rules of the grammar, i.e. only subtrees with the same root can be crossed over, and random generation of trees must follow the derivation rules. A basis function is the leaf nodes (terminal symbols) of the tree; internal nodes (nonterminal symbols) reflect the underlying structure; the tree root is the start symbol.

CAFFEINE (Canonical Functional Form Expressions in Evolution) [11] included a carefully designed grammar specifically for modeling functional forms. The grammar allowed all functional compositions, but in just one canonical form. A CAFFEINE individual consisted of a set of trees. The root node of each tree (basis function) was a product of sums; and the weights on the basis functions could be linearly learned from *y*. In IBMG, however, the usage of the set of basis functions is different: the expressions are simulated in a dynamic fashion, starting from an initial state $z(0)$, and each the change in state dx/dt at time t is a function of the current state $x(t)$ and inputs $u(t)$. Thus, linear weights on the basis functions cannot be determined by linear learning. This means that the form for the root should be more open-ended; it can be either a weighted sum of basis functions or a product of variables and/or nonlinear functions. Once the system is simulated, which computes $z(t)$ for each t, then linear learning can be employed on z and y to determine E and F. Accordingly, the IBMG grammar makes just one adjustment to the CAFFEINE grammar: it adds the first line in the grammar. The IBMG grammar is:

```
ROOTSYM => W + REPADD | REPVC

REPVC => VC | REPVC * REPOP | REPOP

REPOP => REPOP * REPOP | 1OP( W + REPADD ) |

         2OP( 2ARGS ) | ... (3OP, 4OP etc)

2ARGS => W + REPADD, MAYBEW | MAYBEW, W + REPADD

MAYBEW => W | W + REPADD

REPADD => W * REPVC | REPADD + REPADD

2OP    => DIVIDE | MAX | ...

1OP    => INV | LOG10 | ...
```

REPVC and REPOP is a product of variables and/or nonlinear functions. Within each nonlinear function is a weighted sum of basis functions (REPADD). Each basis function can be, once again, a product of variables and/or nonlinear functions. And so on.

IBMG treats basis function operators slightly different than CAFFEINE because expression inputs are more tightly tied to the basis functions. In CAFFEINE the order of the basis functions was not important because the output was a weighted sum. In IBMG, the location i of a basis function describes the state variable $z(i)$ that it updates. With this in mind, basis function operators include: uniform crossover of basis functions from two parents; deleting a random basis function (therefore making its location empty); adding a randomly generated tree as a basis function; copying a subtree from one individual to make a new basis function for another.

The grammar is context-free, with two exceptions for the sake of enhanced search: weights and variable combinations (VCs). At each weight (W) node, a real value is stored in the range [-2*B, +2*B]. During interpretation of the tree the value is transformed into $[-1e+B,-1e-B] \cup [0.0] \cup [1e-B, 1e+B]$. B is user-set, e.g. 10. In this way parameters can take on very small or very large negative or positive values. Cauchy mutation is the operator. Each VC has an accompanying vector that has one integer value per variable as the variable's exponent. There is one variable for each of the p system inputs and one variable for each of the basis functions in the candidate model. For interpretability, only integer exponents are allowed. VC operators include: one point crossover, and randomly adding or subtracting to an exponent value.

VI. Experimental Comparison

A. Experimental Setup

A prototype IBMG system was written in about 2000 lines of Matlab code. The grammar was defined in a separate text file. Run settings were: maximum number of basis functions 10, population size 250, number of generations 1500, maximum tree depth 8, weight setting B=4, complexity measure settings $w_b = 10$, $w_{vc} = 0.25$. All operators had equal probability, except parameter mutation was five times more likely. Single-input operators allowed were: $\sqrt{x}$, $\log_e(x)$, $\log_{10}(x)$, $1/x$, $\text{abs}(x)$, x^2, $\sin(x)$, $\cos(x)$, $\tan(x)$, $\max(0, x)$, and $\min(0,x)$, 2^x, 10^x, where x is an expression. Dual-input operators allowed are x_1+x_2, x_1*x_2, $\max(x_1,x_2)$, $\min(x_1,x_2)$, $\text{power}(x_1,x_2)$, and x_1/x_2. Also, lte(*testExpr*, *condExpr*, *exprIfTestLessThanCond*, *elseExpr*) and lte(*testExpr*, 0, *exprIfTestLessThan0*, *elseExpr*) were used, along with gte. We could have turned off any unwanted rules or operators, e.g. to restrict the search to linear functions or rationals, but we kept the search open-ended. Experiments are on a 3.0 GHz Pentium IV PC running Matlab 6.5 on Red Hat Linux.

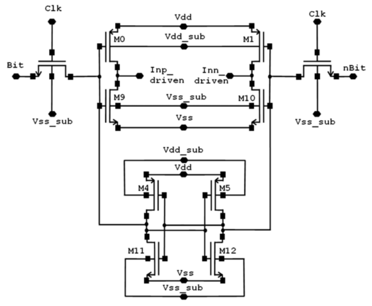

Figure 1: Latch circuit

To test IBMG, we model a strongly nonlinear circuit: a latch used in a DAC system similar to [15], shown in Figure 1. It synchronizes the digital, differential input data at nodes `Bit-nBit` with the clock `Clk`. `Bit`, `nBit` and `Clk` have been passed through buffers. All the transistors except M14 and M13 form a David-Goliath (strong-weak) inverter structure. The David inverters regenerate the signals on the gates of M0/M1/M9/M10 and hold them while the pass transistors M14 and M13 are off. The technology is 0.18μm CMOS. `Vdd`, `Vdd_sub`, `Vss`, and `Vss_sub` are 1.8V, 1.8V, 0.0V, and 0.0V respectively. Figure 2 shows the circuit's input and output waveforms. IBMG's goal is to build a model that produces similar outputs given those same inputs. `Vdd` and `Vss` are also treated as inputs to IBMG. Each waveform had 2001 samples.

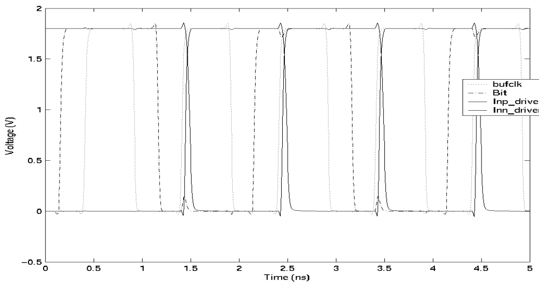

Figure 2: Input and target output waveforms (`nBit`, not shown, is merely the inverse of `Bit`)

B. Model Prediction Results

We ran IBMG to build models for the latch. Runtime was 72 hours (a compiled implementation would be about an order of magnitude faster). Figure 3 shows the best-performing result, which achieved $nmse_{tot}$ of 1.31%. This is a fairly tight fit, especially given that IBMG did not use the circuit's internal state information and instead had to invent its own states and state transition equations. Examining the waveform, we see that the sharp nonlinear transitions are handled quite gracefully, though the model output jumps around somewhat at around 0.5 ns. The output is fairly smooth over time in part thanks to minimization of error of derivatives. Thus, IBMG has accomplished the error-minimization goal.

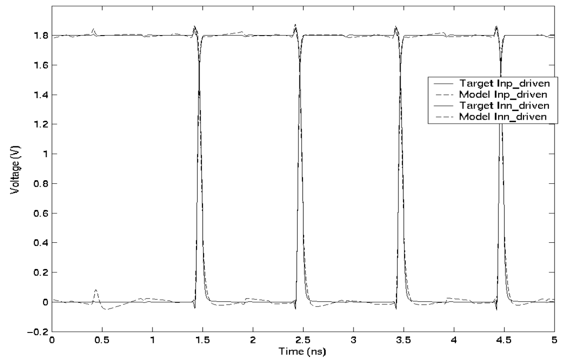

Figure 3: Target output signals and model output signals

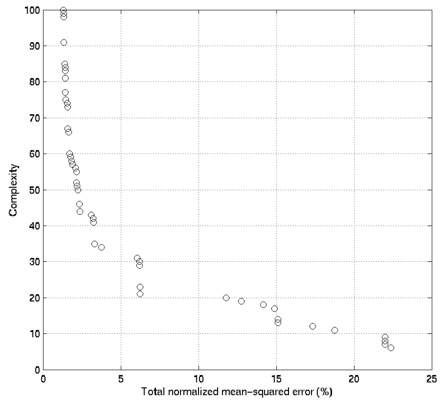

Figure 4: Tradeoff of complexity vs. normalized mean-squared error

% Err	Expression
15.1	$dx_1/dt = nBit$, $dx_2/dt = Bit * x_1$
6.25	$dx_1/dt = - 21.3 - 9.28e-03 * bufclk * x_1 + 1.0e+04 * nBit * bufclk$
3.32	$dx_1/dt = 2.21e-02 - 3.72e-02 * x_1 - 21.8 * Bit*nBit * bufclk$ $dx_2/dt = nBit * bufclk * x_1$, $dx_6/dt = x_1$
1.31	$dx_1/dt = 78.2 + 1.06e-03 * Bit * x_1 - 2.11e-02 * bufclk * x_1 - 4.85 * Bit * nBit * bufclk * x_{10}$ $dx_2/dt = nBit * bufclk * x_1$ $dx_3/dt = x_1$ $dx_4/dt = Bit * nBit * bufclk * x_1 * x_{10}$ $dx_6/dt = Bit * nBit * bufclk * x_1$ $dx_8/dt = Bit * nBit * bufclk$ $dx_9/dt = bufclk * x_1$ $dx_{10}/dt = 25.9 + 1.44e-04 * Bit * x_1 - 1.89e-03 * x_{10}$

Table I: Some behavioral models of the latch generated by IBMG

Figure 4 illustrates the outcome of IBMG's error-control strategy: a set of about 50 behavioral models that collectively trade off model complexity with error. Table I shows some IBMG models. The most complex one, in the bottom row, achieved 1.31% error, yet is readily interpretable. It effectively only has two state variables (x_1 and x_{10}); the other six only act to create nonlinear mappings from x and u to y, which is fine, as section 1 discussed. Interestingly, polynomials almost exclusively dominated the final models, though some expressions had *max* and some intermediate results with error less than 2.0% had used *lte0* and *square*. This is reasonable, as polynomials are among the smoothest, simplest expressions.

Conclusions

We have presented IBMG, an approach to generate behavioral models of nonlinear analog circuits, with the special distinction that the models are compact, *interpretable* expressions that are not restricted to any pre-defined functional templates. The key is the use of a specialized grammar within the context of genetic programming. To test IBMG, we used it to model a strongly nonlinear circuit, namely a latch. IBMG successfully generated compact, interpretable models that trade off complexity with error. Even the best-fit model (1.3% error) was very interpretable, which greatly aids trust.

Acknowledgment

Thanks to J. Deveugele for helping set up the latch problem.

References

[1] G. Gielen, R.A. Rutenbar, "Computer aided design of analog and mixed-signal integrated circuits," Proc. of the IEEE Vol. 88(12), 2000, pp. 1825-1849.

[2] J. Roychowdhury, "Automated macromodel generation for electronic systems," BMAS 2003, San Jose, CA

[3] M. Rewienski, J. White, "A trajectory piecewise-linear approach to model order reduction and fast simulation of nonlinear circuits and micromachined devices," Proc. ICCAD 2001, San Jose, CA, 2001, pp. 252-257

[4] N. Dong, J.S. Roychowdhury, "Piecewise polynomial nonlinear model reduction," DAC 2003: 484-489

[5] J.R. Phillips, J. Afonso, A.L. Oliveira, L.M. Silveira, "Analog macromodeling using kernel methods," ICCAD 2003, pp. 446-453

[6] J.R. Phillips, "A statistical serspective on nonlinear model reduction," BMAS 2003, San Jose, CA

[7] D.E. Root, J. Wood, N. Tufillaro, "New techniques for non-linear behavioral modeling of microwave/RF ICs from simulation and nonlinear microwave methods," DAC 2004

[8] J.R. Koza. Genetic Programming. MIT Press, 1992.

[9] H. Cao, L. Kang, Y. Chen, J. Yu, "Evolutionary Modeling of Ordinary Differential Equations with Genetic Programming," Genetic Programming and Evolvable Machines 1(4), Oct. 2000, pp. 309-337

[10] W. B. Langdon, R. Poli, "Fitness Causes Bloat: Mutation," Lecture Notes in Comp. Sci. 1391, Springer, 1998, pp. 37-48

[11] T. McConaghy, T. Eeckelaert, G. Gielen, "CAFFEINE: Template-free symbolic model generation of analog circuits via canonical form functions and genetic programming", DATE 2005 (not published)

[12] K. Deb, S. Agrawal, A. Pratap, T.A. Meyarivan, "A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II," PPSN VI 2000, pp. 849-858

[13] P. Whigham, "Grammatically-based genetic programming," Workshop on GP: From Theory to Real-World Apps., 1995.

[14] J.R. Phillips, "Projection-based approaches for model reduction of weakly nonlinear, time-varying systems," IEEE Trans CAD, Feb. 2003, pp. 453

[15] J. Deveugele, M. Steyaert, "A 10b 250MS/s binary-weighted current-steering DAC," ISSCC 2004