# New Generation Verilog-A Model Development Tools: VAPP and VALint

*A. Gokcen Mahmutoglu*
*Xufeng Wang*
*Jaijeet Roychowdhury*

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2018-89
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-89.html

July 4, 2018

# New Generation Verilog-A Model Development Tools: VAPP and VALint

A. Gokcen Mahmutoglu, *Member, IEEE*, Xufeng Wang, *Member, IEEE* and Jaijeet Roychowdhury, *Fellow, IEEE*

***Abstract*—We present software tools, VAPP and VALint, for the development of new Verilog-A compact models and also for applications involving existing models. VAPP, the Berkeley Verilog-A Parser and Processor, translates Verilog-A device models into executable and accessible model code. VALint is a graphical code quality checking tool. By virtue of its intuitive syntax for creating compact device models, Verilog-A has come to be used as the standard compact modeling language in the electrical engineering community. However, the high-level language constructs of Verilog-A necessitate the translation of device model code into a lower-level model description format before it can be used in simulations. VAPP runs in MATLAB/Octave, takes a Verilog-A model as input and, by default, generates executable model code in the open ModSpec format complete with symbolically computed derivatives. VAPP features a modular software architecture which can be easily modified and extended to be used with different model description formats and target programming languages. Together with the Berkeley Model and Algorithm Prototyping Platform (MAPP), VAPP offers a powerful framework for testing, debugging and analyzing compact device models. VALint assists model developers in writing clean Verilog-A code by checking models for common mistakes and bad Verilog-A practices. VALint implements rules for best Verilog-A modeling practices accumulated over the years by leading industry experts. VAPP and VALint are freely available and released as open source code.**

***Index Terms*—Compact device modeling, Verilog-A.**

## I. Introduction

Circuit simulators use compact device models to mimic the behavior of physical devices in simulations. Each simulator has a different format to describe device models. The details of this format depend on the simulator architecture and its data structures. Consequently, device model developers face the problem of having to provide multiple implementations of the same model if they want to support different simulators. Over the years, the Verilog-A modeling language has come to be used as a remedy for this problem and provided a common programming language to the compact device modeling community [1].

Today, most compact semiconductor device models are written using Verilog-A, *e.g.*, [2], [3], [4], and most circuit simulators provide tools to convert Verilog-A models into their internal model description formats. These tools are known as Verilog-A compilers [5]. On the one hand, Verilog-A models cannot be tested, debugged or used in simulations unless they are translated via a compiler. On the other hand, the translated lower-level model code is usually not accessible to the user because of the opaque and closed nature of commercial simulators. This makes compact model development an arduous process since developers lack the necessary testing and debugging tools.

In this paper, we present open source tools aimed both at the developers and the users of compact models. These tools will enable model developers to create better models in a streamlined manner. Moreover, users of compact models will be able to translate, examine and execute complicated Verilog-A model code. The first tool we present is the Berkeley Verilog-A Parser and Processor (VAPP), an open source, modular Verilog-A compiler. The second one is VALint, a model quality checker built on VAPP's infrastructure. Both of these tools are integrated into the larger framework of the Berkeley Model and Algorithm Prototyping Platform (MAPP) [6].

VAPP is completely written from scratch in MATLAB[1] and seamlessly integrates with MAPP. The design of the software is object oriented and modular to facilitate easy extensions. VAPP takes a Verilog-A file as input and prints out a lower-level model description format which can be used in simulations directly. The default output format of VAPP is ModSpec [7], the executable model description format used in MAPP. Once a Verilog-A model is converted into a ModSpec file, it can be executed directly, *e.g.*, in MATLAB/Octave, to produce test data or to visualize the characteristic curves of the device. Unlike its predecessor, ADMS [8], using VAPP does not require creating complicated XML code generator specifications. Users with basic programming knowledge can write their own backends composed of simple but powerful print functions to create different output formats or different target languages.

VALint utilizes the powerful visitor design pattern of VAPP to process Verilog-A input files and checks them for mistakes. It implements a set of rules regarding best practices for Verilog-A models laid down by industry experts [5], [9], [10]. Through its graphical user interface, VALint visually marks problem spots in model code and provides suggestions to eliminate them.

In the remainder of this paper, we present details about the software architecture and the internal data structures of VAPP. We provide guidelines on how to implement additional features such as a new backend. We demonstrate, with examples, how VAPP and ModSpec can be powerful tools in identifying regions where a model breaks down. We also provide sample use cases of VALint and point out how it can help to write cleaner Verilog-A models.

## II. VAPP—The Berkeley Verilog-A Parser and Processor

VAPP provides a single-command user interface. By default, when called with the filename of the Verilog-A model as an argument, VAPP will print its output to a file with the same name as the main Verilog-A module. For example, the command

```
vapp('bsim6.1.1.va');
```

will produce a file with the name `bsim6.m`, `bsim6.cpp` *etc.* depending on the target language. The output filename can, of course, be changed using the options provided by the command-line interface along with various other settings.

Figure 1 shows the fundamental steps in VAPP's model translation process. First, the input Verilog-A file is parsed by VAPP's *frontend*. The resulting Abstract Syntax Tree (AST) contains syntactic information about the input file such as nodes/branches, mathematical operations, contributions and user defined functions. The AST serves as an input to VAPP's core, which, among other tasks, defines model inputs/outputs, creates potentials/flows (voltages/currents), discovers the dependency structure of the model variables/outputs, and

A. Gokcen Mahmutoglu and Jaijeet Roychowdhury are with the EECS department, University of California, Berkeley (amahmutoglu@berkeley.edu, jr@berkeley.edu). Xufeng Wang is with the ECE deparment, Purdue University (wang159@purdue.edu).

[1] A separate, Octave compatible version is also available.

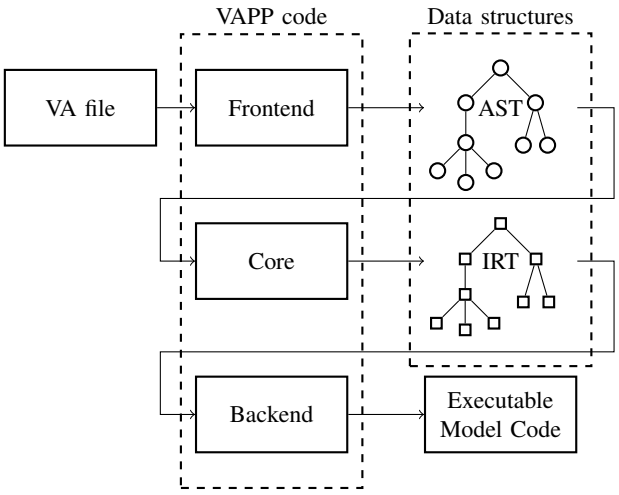



Fig. 1: VAPP's operation steps. A Verilog-A file is supplied as input and a lower-level model code is produced as output. The internal data structures of VAPP are marked with AST (abstract syntax tree) and IRT (intermediate representation tree) labels.

computes derivatives. This entire information is encoded in another internal data structure called the Intermediate Representation Tree (IRT). Finally, a backend is used to generate a printout of the IRT in the desired programming language and the model description format. In the remaining part of this section, we will describe how the individual parts of VAPP function and how their operation can be customized to suit different data processing needs and output formats.

After parsing the input file, VAPP's frontend produces an AST. The AST is a representation of the Verilog-A expressions in the form of a Directed Acyclic Graph (DAG). An example of this is given in Figure 2(a) for the following Verilog-A contribution line describing a resistor and inductor pair in series.

```
V(p,n) <+ R*I(p,n) + L*ddt(I(p,n));
```

Because of the relatively simple nature of the AST, VAPP represents every node in the tree with the same class, `VAPP_AST_Node`. The type and attributes of each node is determined by the values of the class properties (data members).

Figure 2(b) shows the IRT representation of the AST in Figure 2(a). The IRT encodes much more complex information than the AST. Every node in the IRT is an instance of a special class. A few examples of these classes are `IrNodeContribution`, `IrNodePotentialFlow` and `IrNodeParameter`. As opposed to the AST, the information in these classes does not only represent the Verilog-A expressions in the model file but contains rich data needed to analyze and reformulate a device model. Examples of these data fields are shown in Figure 2(b). For instance, the voltage on the LHS (`V(p,n)`), and the current on the RHS of the contribution (`I(p,n)`) are marked as outputs and inputs, respectively. Moreover, objects such as nodes and branches are represented with their own classes. The `(p,n)` branch in Figure 2(b), for example, is represented with an `MsBranch` object which is part of an `MsNetwork` object. Both the IRT objects (prefixed with `IrNode`) and model specification objects (prefixed with `Ms`) provide diverse methods for data retrieval and analysis. Developers of new backends are expected to use these properties and methods to create their own output formats.

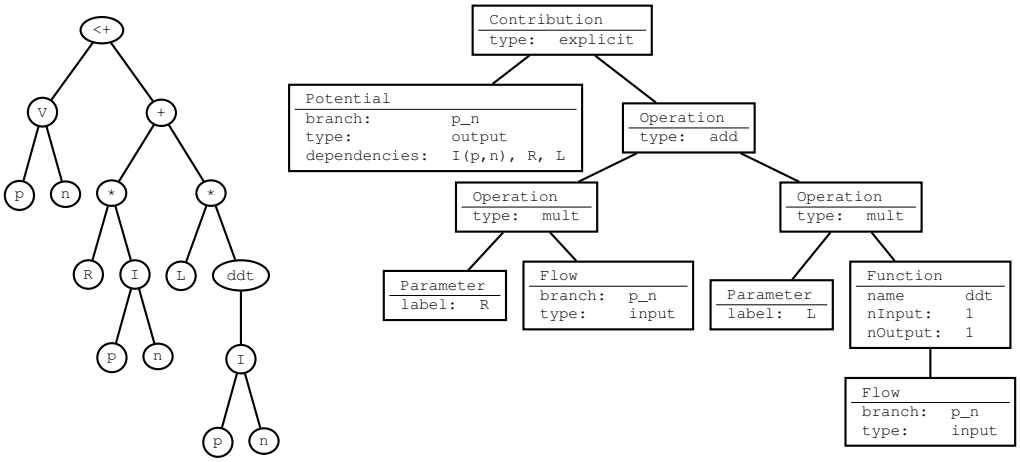
VAPP implements the *visitor design pattern* for easy traversal and manipulation of ASTs and IRTs [11]. VAPP's internal computations are implemented as individual visitor classes. For example, the dependency structure of parameters, variables, potentials and flows is created using the `IrVisitorGenerateDependecy` class. Visitor classes that traverse the IRT implement a visit method for each different `IrNode` class while AST visitors implement a visit method for each type of `VAPP_AST_Node` object. The name of the visit method determines for which type of node it will get called. These method names are defined in the `AstVisitor` and `IrVisitor` interface classes. An example of a visitor class is given in Listing 1. This `AstVisitor` class counts the number of arithmetic operations and function calls in a model file to estimate the runtime of a device evaluation in simulations. Developers can implement their own visitor classes and use them on ASTs and IRTs generated by VAPP.

An important capability of VAPP is provided by the visitor class `IrVisitorGenerateDerivative`. Simulators require the derivatives of model outputs with respect to their inputs—also known as the model Jacobian. These can be computed by the simulator using an automatic differentiation technique as in MAPP [6] and Xyce [12]. However this method generally slows down device evaluations because it relies on techniques such as operator overloading which introduce additional overhead to the computations. The alternative to automatic differentiation is to include hard-coded derivatives into the model itself. The `IrVisitorGenerateDerivative` class of VAPP traverses the IRT and computes derivatives symbolically. The IRT is then extended with additional nodes for the derivative computations. These additional nodes become a part of the IRT and can be treated as any other parts of the tree, *e.g.*, using visitors. This means that VAPP offers the possibility of computing *higher order derivatives* by the repeated application of `IrVisitorGenerateDerivative`.

The *raison d'etre* of VAPP is to translate a Verilog-A model into a model description format native to a specific simulator. The default output format of VAPP is ModSpec, the model specification format of MAPP [6], [7]. However, VAPP's modular structure makes it easy to specify rules for other output formats. In order to create a new output format, one has to implement a single class (a backend) containing print functions for each node type in the IRT. An example of such a function is given in Listing 2. This function is part of VAPP's default backend and facilitates the printing of an if/else statement in MATLAB syntax. In lines 6 and 8, it calls the print methods of its first two children (the condition and the then-statement) and also prints the else-statement if it exists (lines 12-18). It is easy to see how this method can be modified to use a different target language, *e.g.*, C++. In the same spirit, the entire backend can be modified to generate model code in a different model description format other than ModSpec. The open source nature of VAPP enables the implementation of different backends for various simulators.

## III. VALint: A Graphical Code Quality Checking Tool

VALint is a Verilog-A code quality checking tool. It identifies bad practices, common mistakes, programming pitfalls, and inefficiencies using VAPP's AST and IRT data structures. VALint analyzes device models using a list of best modeling practices inspired by several publications on this subject, *e.g.*, [5], [9], [10], [13], [14]. These rules identify problems in model code which are not usually covered by Verilog-A compilers. These undesired patterns in Verilog-A code do not usually interrupt the compilation process, but they can cause various problems ranging from performance degradation to program malfunction. A prominent example of an unwanted Verilog-A programming practice is the use of *hidden states* [10]. Hidden states are usually caused by local variables in the Verilog-A code which might not get initialized under certain inputs to the model function, *e.g.*, because of conditional statements [**?**]. Hidden states can cause issues




(a) (b)

Fig. 2: (a) The Abstract Syntax Tree (AST) representation of the Verilog-A expression, `V(p,n) <+ R*I(p,n) + L*ddt(I(p,n))`. (b) The Intermediate Representation Tree (IRT) version of the same expression. The nodes of the IRT contain various information fields and references to objects like branches and potentials/flows.

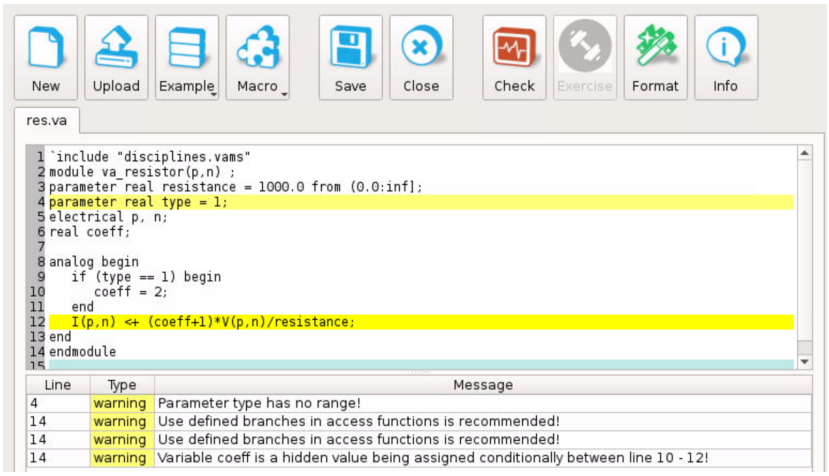



Fig. 3: Web-deployed VALint on nanoHUB.org. The GUI is accessible entirely through a web browser without any installation needed.

in simulations such as convergence problems or they can cause some analysis types, such as PSS, to completely fail. VALint

The initial core VALint code was based on ADMS. An alpha-testing release of this version was first published on nanoHUB.org. However, the development of the ADMS based design quickly ran into various issues. The reliance of ADMS on the aging lexer and parser toolchain of Flex and Bison made both the development and the installation processes a challenge. The static and limited parsing scheme prevented the implementation of several new VALint rules in a quick manner. For example, checking for the hidden states requires a vigorous back-tracing of variable dependencies which has proved to be difficult to do under the Bison parsing framework. These and several similar reasons necessitated the migration of the VALint core to VAPP. This VAPP based built enabled VALint to be fully integrated with MAPP and its simulation/analysis tools.

VALint provides a command-line user interface that runs entirely on MATLAB. It also features a QT based Graphical User Interface (GUI) to compliment the command-line version. The GUI version of VALint can either be installed as a standalone program or it can be run on the cloud through a web browser[2]. Figure 3 shows a snapshot of VALint's cloud based GUI. The GUI supports uploading Verilog-A file from the local machine, and the evaluation of the code is performed remotely on the cloud. After the evaluation, the output is displayed in the GUI with parts of the model code highlighted. VALint provides detailed information about the errors/warnings/-suggestions for these lines in a bottom panel. Various convenience features are also included such as sorting the output messages.

The sample Verilog-A model in Figure 3 implements a simple resistor with a resistance value that depends on the parameter `type` (this simple example was chosen because it fits nicely into the figure estate). The code is syntactically valid and runs without errors in commercial simulators. However, it contains various poor programming practices which are detected and highlighted by VALint. The GUI snapshot in Figure 3 displays four warnings: a parameter without a predefined range, two instances of node labels in an access function where a branch would be more appropriate, and a hidden state.

## IV. Example Applications for Model Development and Analysis

As we have discussed in Section II, VAPP is versatile in analyzing device models and easily extendible to produce model formats native

[2]This service is offered through nanoHUB.org.

```
1  classdef AstVisitorOpCounter < AstVisitor
2  % Count the number of arithmetic operations
3    properties
4      nAdd  = 0;  % number of additions
5      nMult = 0; % number of multiplications
6      nFunc = 0; % number of function calls
7    end
8
9    methods
10     function out = visitOp(this, opNode)
11       % Whenever we visit an operation node
12       % do the following
13       opType = opNode.get_attr('op');
14       switch opType
15         case {'+', '-'}
16           this.nAdd = this.nAdd + 1;
17         case {'*', '/'}
18           this.nMult = this.nMult + 1;
19         end
20        out = true; % continue to child nodes
21     end
22
23     function out = visitFunc(this, funcNode)
24     % Whenever we visit a function call:
25       this.nFunc = this.nFunc + 1;
26        out = true; % continue to child nodes
27     end
28   end
29 end
```

Listing 1: Example usage of VAPP's visitor pattern to estimate the runtime of a model by counting arithmetic operations and function calls. The `visitOp` method defined in line 10 is called by VAPP whenever it encounters an AST node of type `op`. Similarly, the `visitFunc` method (line 23) increments the `nFunc` property of the class whenever a function call is encountered.

```
1  function [outStr, printSub] = printIfElse(node)
2    % don't continue printing the children
3    printSub = false;
4
5    % the condition string
6    condStr = node.getChild(1).sprintAll();
7    % the then string
8    thenStr = node.getChild(2).sprintAll();
9
10   outStr = ['if', condStr, '\n',...
11             node.sprintfIndent(thenStr)];
12   if node.nChild > 2 % is there an else part?
13     elseStr = node.getChild(3).sprintAll();
14     if isempty(elseStr) == false
15       outStr = [outStr, 'else\n',...
16                 node.sprintfIndent(elseStr)];
17     end
18   end
19   outStr = [outStr, 'end\n'];
20 end
```

Listing 2: Example of a VAPP backend print function. During the generation of the executable model code, VAPP calls this method when it encounters an `if/else` block in the IRT.

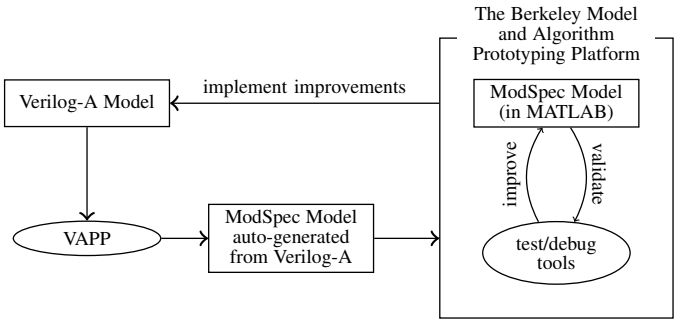



Fig. 4: Compact model development flow using VAPP and MAPP. The development starts at the upper left corner, with a prototype Verilog-A model. This prototype is translated using VAPP and the model is iteratively refined in MAPP.

to different simulators. However, when viewed in the larger framework of the Berkeley Model and Algorithm Prototyping Platform, VAPP fulfills a more fundamental function vis-à-vis the development process of new device models. In this context, the appeal of VAPP is that Verilog-A device models can immediately be converted into ModSpec models which are directly executable in MATLAB/Octave using MAPP. This means, that one can easily produce characteristic curves of devices and spot problematic points. One can also test and debug device models in a user friendly programming environment.

Figure 4 shows the principal model development flow using MAPP and VAPP. The development of a new model starts in the upper left corner by coding a prototype in Verilog-A. This Verilog-A model is then translated using VAPP into ModSpec. The ModSpec model can now be used as an input to MAPP's model analysis tools. One of these tools is the *model exerciser* which can evaluate and plot the model equations and model outputs. It can also compute DC solutions for models with internal unknowns and plot characteristic curves of currents and voltages. Utilizing MAPP's model development tools and algorithms, the ModSpec model is then validated in a testing/debugging loop where the model is iteratively improved. These improvements are then implemented in the original Verilog-A model. The above procedure can be applied as many times as necessary in order to arrive at a model that can be reliably used in simulations.

Besides developing new models from scratch, VAPP can also be used for spotting problems in existing models. One of the ways glitches in device models manifest themselves is as convergence problems in simulations. In those cases, models can be examined using the process in Figure 4 in order to pinpoint the reasons for the problem. Figures 5 and 6 show examples of the application of this process in a real-life research setting. Figure 5 shows a discontinuity in the drain current of the BSIM3 model [15]. This discontinuity was discovered when working with this model in a project on speeding up the device evaluation time [16]. Similarly, Figure 6 shows a discontinuity in the derivative of the drain current with respect to the drain voltage in the MVS model [17]. Both of these problems were discovered using the VAPP/MAPP toolchain.

## V. Summary and Conclusion

We have presented two modern software tools for compact model development in Verilog-A: VAPP and VALint. VAPP can translate Verilog-A models into executable code. The default output format of VAPP is ModSpec which can be imported into MAPP [6]. Device models can then be tested and debugged via MAPP's simulation and analysis tools. Using VAPP, one can also import industry standard device models into MAPP and utilize them in circuit design activities. The list of popular compact semiconductor models which have been translated into ModSpec using VAPP includes BSIM3 [15], BSIM4 [18], BSIM6 [2], R3 [4], PSP [3], HiSIM [19], Mextram [20] and others. VALint offers a GUI that can load Verilog-A models and highlight parts of the model code that can cause problems in simulations. We believe that using VAPP and VALint in conjunction

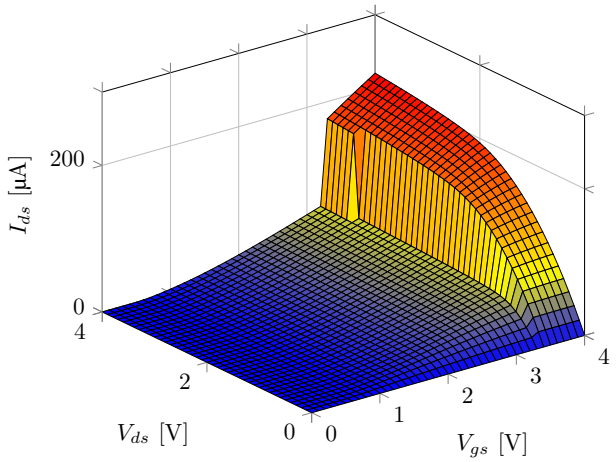



Fig. 5: The discontinuity in the drain current of the BSIM3 model.

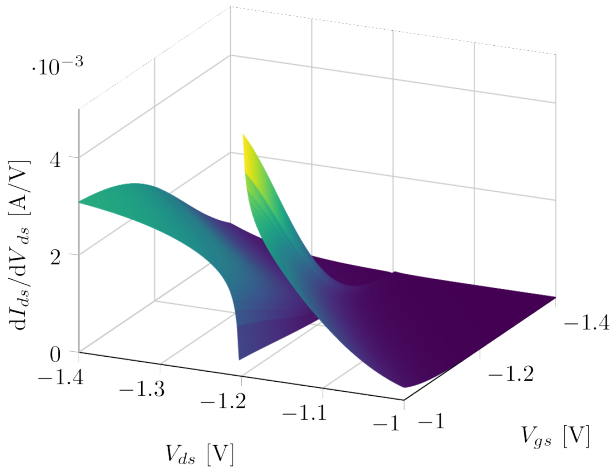



Fig. 6: The discontinuity in the drain current derivative of the MVS v1.0.1 model.

with MAPP will enable compact model developers to produce higher quality models with less effort than the conventional methods.

## References


[1] L. Lemaitre, G. Coram, C. C. McAndrew, and K. Kundert, "Extensions to Verilog-A to support compact device modeling," in *Behavioral Modeling and Simulation, 2003. BMAS 2003. Proceedings of the 2003 International Workshop on*. IEEE, 2003, pp. 134–138.

[2] Y. S. Chauhan, S. Venugopalan, M. A. Chalkiadaki, M. A. U. Karim, H. Agarwal, S. Khandelwal, N. Paydavosi, J. P. Duarte, C. C. Enz, A. M. Niknejad, and C. Hu, "BSIM6: Analog and RF Compact Model for Bulk MOSFET," *IEEE Transactions on Electron Devices*, vol. 61, no. 2, pp. 234–244, Feb. 2014.

[3] G. Gildenblat, X. Li, W. Wu, H. Wang, A. Jha, R. van Langevelde, G. Smit, A. Scholten, and D. Klaassen, "PSP: An Advanced Surface-Potential-Based MOSFET Model for Circuit Simulation," *IEEE Transactions on Electron Devices*, vol. 53, no. 9, pp. 1979 –1993, Sep. 2006.

[4] R. V. Booth and C. C. McAndrew, "A 3-terminal model for diffused and ion-implanted resistors," *IEEE Transactions on Electron Devices*, vol. 44, no. 5, pp. 809–814, 1997.

[5] G. Coram, "How to (and how not to) write a compact model in Verilog-A," in *Behavioral Modeling and Simulation, 2004. BMAS 2004. Proceedings of the 2004 International Workshop on*. IEEE, 2004, pp. 97–106.

[6] T. Wang, A. V. Karthik, B. Wu, J. Yao, and J. Roychowdhury, "MAPP: The Berkeley Model and Algorithm Prototyping Platform," in *2015 IEEE Custom Integrated Circuits Conference (CICC)*, Sep. 2015, pp. 1–8.

[7] D. Amsallem and J. Roychowdhury, "ModSpec: An open, flexible specification framework for multi-domain device modelling," in *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on*. IEEE, 2011, pp. 367–374.

[8] L. Lemaitre, C. C. McAndrew, and S. Hamm, "ADMS-automatic device model synthesizer," in *Custom Integrated Circuits Conference, 2002. Proceedings of the IEEE 2002*. IEEE, 2002, pp. 27–30.

[9] A. G. Mahmutoglu, T. Wang, A. Gupta, and J. Roychowdhury, "Well-Posed Device Models for Electrical Circuit Simulation," 2017. [Online]. Available: https://nanohub.org/resources/26199

[10] C.C. McAndrew, G.J. Coram, K.K. Gullapalli, J.R. Jones, L.W. Nagel, A.S. Roy, J. Roychowdhury, A.J. Scholten, Geert D.J. Smit, X. Wang and S. Yoshitomi, "Best Practices for Compact Modeling in Verilog-A," *IEEE J. Electron Dev. Soc.*, vol. 3, no. 5, pp. 383–396, Sep. 2015.

[11] J. Palsberg and C. B. Jay, "The essence of the Visitor pattern," in *Computer Software and Applications Conference, 1998. COMPSAC '98. Proceedings. The Twenty-Second Annual International*, Aug. 1998, pp. 9–15.

[12] E. Keiter, T. Mei, T. Russo, R. Schiek, P. Sholander, H. Thornquist, J. Verley and D. Baur, "Xyce Parallel Electronic Simulator Reference Guide Version 6.4." Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2015.

[13] G. Coram and M. Ding, "Recent achievements in Verilog-A compact modeling," in *MOS-AK Workshop*, Baltimore, MD, USA, Dec. 2009.

[14] G. Coram and C. C. McAndrew, "Verilog-A for compact modeling: Best practices for high-quality model authoring," in *Compact RF Modeling Workshop*, Santa Barbara, CA, USA, Sep. 2005.

[15] P. Ko, J. Huang, Z. Liu, and C. Hu, "BSIM3 for analog and digital circuit simulation," in *Proceedings of the IEEE Symposium on VLSI Technology CAD*, 1993, pp. 400–429.

[16] A. Gupta, T. Wang, A. G. Mahmutoglu, and J. Roychowdhury, "STEAM: Spline-based tables for efficient and accurate device modelling," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2017, pp. 463–468.

[17] S. Rakheja and D. Antoniadis, "MVS Nanotransistor Model (Silicon)," https://nanohub.org/publications/15, Oct 2014.

[18] N. Paydavosi, T. H. Morshed, D. D. Lu, W. M. Yang, M. V. Dunga, X. J. Xi, J. He, W. Liu, M. C. Kanyu, X. Jin *et al.*, "BSIM4v4.8.0 MOSFET Model."

[19] M. Suetake, K. Suematsu, H. Nagakura, M. Miura-Mattausch, H. J. Mattausch, S. Kumashiro, T. Yamaguchi, S. Odanaka, and N. Nakayama, "HiSIM: a drift-diffusion-based advanced MOSFET model for circuit simulation with easy parameter extraction," in *2000 International Conference on Simulation Semiconductor Processes and Devices (Cat. No.00TH8502)*, 2000, pp. 261–264.

[20] R. Van der Toorn, J. C. J. Paasschens, and W. J. Kloosterman, "The Mextram bipolar transistor model," *Delft University of Technology, Technical report*, 2008.