Proceedings of the 1996 Winter Simulation Conference
ed. J. M. Charnes, D. J. Morrice, D. T. Brunner, and J. J. Swain

EFFICIENT SIMULATION MODEL GENERATION USING AUTOMATIC PROGRAMMING TECHNIQUES

Jae-Wook Lee

Department of Electrical Engineering
Yonsei University
Seoul, Korea

Sungho Kang

Department of Electrical Engineering
Yonsei University
Seoul, Korea

ABSTRACT

Generating simulation models is a knowledge intensive, time consuming, and error-prone task in implementing a simulator. The main purpose of this research was to find an easy, fast, and reliable way to generate simulation models and model library. To solve this problem, the Automatic Element Routine Generation System (AERO) is developed as an efficient way for automatic model development using domain specific automatic programming techniques. Behavioral and structural models are generated from Boolean equations, truth tables, HDL descriptions, schematic diagrams, or incomplete specifications. Results show that the system could greatly reduce the cost of simulation model generation for CAD systems and, consequently, reduce the design cycle considerably.

1 INTRODUCTION

It is clear that simulation, at various levels, is an essential and commonly used method for verifying the design of digital systems. A simulation model is a program code which represents the functional behavior of an element and is written in a simulator's target language. Simulation models are created and added into a simulator library. The performance of a simulator highly depends on the number of models in the model library and the efficiency of the models. Since a simulator can only simulate systems constructed of the primitives in the model library, if there are not sufficient models, a large system may not be handled by the simulator. Also, if the model is written in an inefficient way, the speed of the simulator is slow.

Therefore, it is very important to find an effective way for developing a model library as well as for generating efficient models. Until recently, simulation models were developed through a knowledge intensive design process which is time consuming and error prone. Additionally, it is not easy to predict the performance of functional elements. To overcome these difficulties, an automatic way of model generation is required. Over the life time of a simulation system, the development of the simulation model library is the largest programming effort. Therefore, automatic model generation is clearly necessary and very significant, since it has the potential of saving hundreds of man-years of code development.

The concept of automatic code generation of functional models was initially introduced in (Szygenda 1979). Several attempts (Han 1991, Yang 1991) have been made to apply domain specific programming techniques (Barstow 1985) in the automatic model generation area, with great ease, accuracy, and efficiency. Domain specific automatic programming techniques allow a user to describe programs using informal and imprecise terms of domain, and produce efficient and reliable programs.

In this research, to eliminate inefficiency of the previous works, a new simulation model generation system, called AERO (Automatic Element Routine Generation System), is developed. Also, the AERO can generate simulation models from various descriptions for various simulation algorithms such as logic, fault, error simulators. This system can be used either when developing new simulators or for upgrading existing simulators, as a model generation tool.

2 CONFIGURATION OF AERO

To implement efficient domain specific automatic programming techniques, the following must be carefully considered. Firstly, knowledge of the application domain must be kept in some form of rule base to guide the program synthesis. It can be classified into domain independent knowledge base about implementing code syntax and domain specific knowledge base about application domain. Secondly, the system must provide the efficient user interface to get specifications and knowledge. Thirdly, in order to handle the users' incomplete or imprecise specifications, the system must be capable of figuring out

the users' intention.

To generate simulation models effectively, the following must be considered. Firstly, the system must generate high performance models since models affect the overall performance of a simulator. Secondly, the system must verify generated models, since the correctness of the models means the accuracy of a simulator. Thirdly, the system must generate models which can easily interface with various simulators, with minimal modification. Also, the system should generate concise models, in order to minimize the usage of memory. Finally, the generated models for sequential devices must have some mechanisms to allocate data for each instance of the element, including internal memory states.

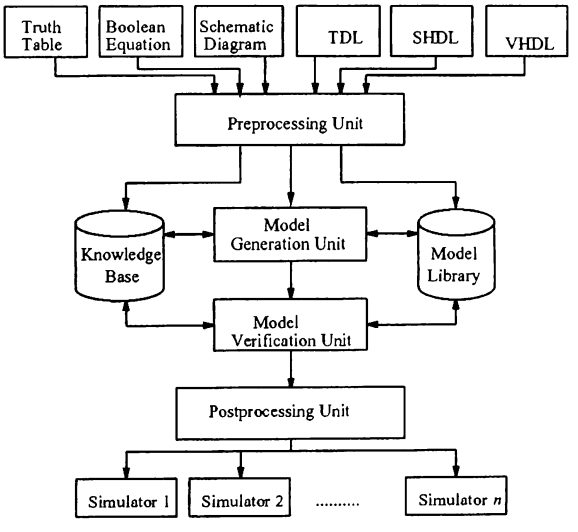

Figure 1: Configuration of the AERO

The AERO (Automatic Element Routine Generation System) can automatically generate models from various descriptions, using domain specific automatic programming techniques. As shown in Figure 1, the AERO consists of several subsystems, i.e., Preprocessing Unit, Automatic Code Generation Unit, Model Verification Unit, Postprocessing Unit, Knowledge Base and Model Library. This system can generate models written in the C programming language from truth tables, boolean equations, schematics, and hardware description languages.

3 PREPROCESSING UNIT

Figure 2 shows a block diagram of the Preprocessing Unit. The Preprocessing Unit includes parsers and translators which have interactive user interface facilities.

From the schematic description, a net list is generated with the help of the OCT environment (Spickelmier 1989). A netlist can also be generated from the hardware description language, Simulation Automation System (SAS) Hardware Description Language (SHDL) (Kang 1991), Tegas Description Language (TDL), and VHDL. These descriptions are parsed, and passed onto the Netlist Generator, which generates the levelized netlists. SHDL and VHDL are also used to describe behavioral domain models. Parsed behavioral descriptions include control flows and equations.

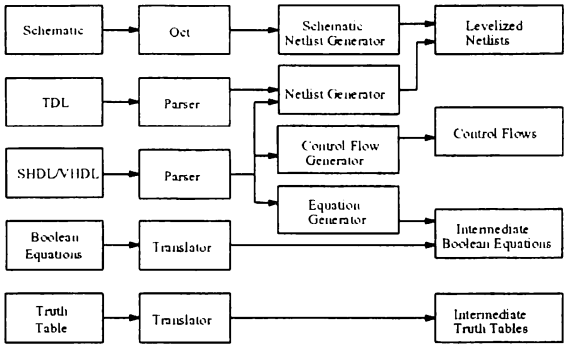

Figure 2: Preprocessing Unit

When truth table descriptions, which are one of the easiest and most natural means to describe a reasonable size circuit, are entered, they are translated into internal data structures. If the Boolean equations, or the gate implementation, of a circuit is known, it is convenient to use equations to describe the circuit. These equations are translated into internal data structures.

Interactive user interface displays the existing models which can be used to construct new models, or to store the generated models and the corresponding model specification into the model library. By grouping similar models into the same class, users can derive the characteristics and transformation rules and attach them onto this class. This knowledge will be used by the incomplete information handler and the input verifier to complete the missing part of the input specification and to perform the consistency check. The series of questions are shown in Figure 3.

4 MODEL GENERATION UNIT

Using the information from the Preprocessing Unit, the Model Generation Unit (MGU) synthesizes models. From the synthesized models, it can generate 'C' models according to user's command. The structure of the MGU is shown in Figure 4.

The general simulation models can be classified into structural and behavioral models. A structural

1. What's the model group name?
2. What's the model name?
3. How many states are desired for this model? [2-10]
4. Do you want to specify bit representations?
5. What's the type of your input specification? [a-h]
   (a) truth table (b) boolean equations
   (c) schematics (d) TDL
   (e) behavioral SHDL (f) behavioral VHDL
   (g) structural SHDL (h) structural VHDL

Figure 3: Questionnaire of Interactive User Interface

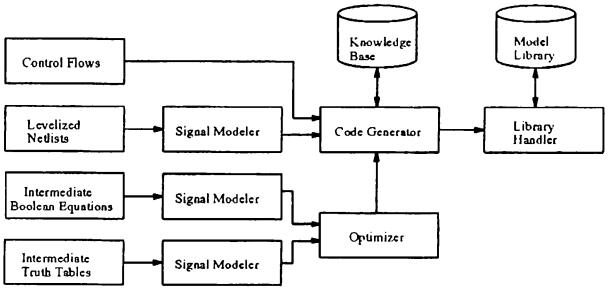

Figure 4: Model Generation Unit

model describes a circuit by connecting low level primitives. A behavioral model is used to describe the functional behavior of a circuit. This system can generate both structural and behavioral models. The simulation model consists of entity declarations, architectural bodies, and configuration declarations. An entity represents a part of a well-defined hardware design. An architecture specifies the relationship between the inputs and the outputs of a design entity, and a configuration specifies the binding of components to entities. With the information about input and output signal names, entity parts can be completed.

If the description is in the form of truth tables or Boolean equations, there are no primitives in the descriptions. However, 2 input AND, 2 input OR, and NOT gates are used to complete an 'architecture' part of structural C models. If the description is in the form of schematics or HDLs, then the information about primitives used in the descriptions is used for the structural models. The remainder of 'architecture' part is filled out using net list information, control flow information, and equation information which is synthesized using domain rules and given specifications.

According to the bit representations of a specific simulator, a signal modeler generates an enhanced table or equations. The signal modeler is used to dynamically customize AERO according to the user specified logic values and bit representations, so that the AERO process becomes very flexible in terms of generating models, which comply with various signal models. For any k-valued signal model, k distinct logic values can be represented in the computer by sets of bits with at least $\log_2 k$ bits in the set. For example, simulators like TEGAS use two word representations for three value simulation. Hence, two variables are used for each variable defined in the original truth table. Let us assume that we want to generate a simulation model for a two input AND gate, for three value logic; where the three state values are L, H, and X, for low signal value, high signal value and unknown value, respectively. The signal modeler may assign; bit0 = 0 and bit1 = 0 for L, bit0 = 0 and bit1 = 1 for H, and bit0 = 1 and bit1 = 1 for X. A C model based on the above bit assignments is shown in Figure 5.

```
AND2(i, o)
int i[2][2], o[1][2];
{
    int a0, a1;
    int b0, b1;
    int c0, c1;

    a0 = i[0][0];
    a1 = i[0][1];
    b0 = i[1][0];
    b1 = i[1][1];

    c0 = (a0&b0);
    c1 = ((a1&b1)|(a1&b0)|(a0&b1));

    o[0][0] = c0;
    o[0][1] = c1;
}
```

Figure 5: 2 Input AND Model for 3 Values

For more accurate simulation, 5 logic values (0, 1, U, D, E) are usually used. To represent 5 value, 3 bits are used, where logic L is represented by bit pattern 000, logic H is represented by bit pattern 110, the state U (for signal going-up) is represented by bit pattern 010, the state D (for signal going-down) is represented by bit pattern 100, and the state E (for error or unknown value) is represented by bit patterns 001, 011, 101, or 111. However, the encoding for the given value is not unique and the users should be able to decide on their own selections. A C model based on the above bit assignments is shown in Figure 6.

As an example, consider path delay fault sim-

```
OR2(i, o)
int i[2][3], o[1][3];
{
    int a0, a1, a2;
    int b0, b1, b2;
    int c0, c1, c2;

    a0 = i[0][0];
    a1 = i[0][1];
    a2 = i[0][2];
    b0 = i[1][0];
    b1 = i[1][1];
    b2 = i[1][2];

    c0 = (a0|b0);
    c1 = (a1|b1);
    c2 = ((¬a0&b2)|(¬a1&b2)|(a2&¬b0)|(a2&¬b1)
      (a2&b2)|(a0&¬a1&¬b0&b1)|(¬a0&a1&b0&¬b1));
    o[0][0] = c0;
    o[0][1] = c1;
    o[0][2] = c2;
}
```

Figure 6: 2 input OR Model for 5 Values

```
MUX21(i, o)
int i[3][2], o[1][2] ;
{
   int a0, a1, b0, b1, c0, c1, d0, d1 ;

   a0 = i[0][0] ;
   a1 = i[0][1] ;
   b0 = i[1][0] ;
   b1 = i[1][1] ;
   c0 = i[2][0] ;
   c1 = i[2][1] ;

   d0 = (¬c0&a0)|(c0&b0)|
      (c1&(a0|b0|(¬a1&b1)|(a1&¬b1)));
   d1 = (¬c0&a1)|(c0&b1)|
      (c1&(a1|b1|(¬a0&b0)|(a1&¬b0)));

   o[0][0] = d0 ;
   o[0][1] = d1 ;
}
```

Figure 7: 2-to-1 MUX for Delay Fault Simulation

ulation (Kang 1994) using 4 values. Logic L is represented by bit pattern 00, logic H is represented by bit pattern 10, logic X is represented by bit pattern 11, and logic Z is represented by bit pattern 01. The evaluation routine for 2-to-1 MUX is shown in Figure 7.

When equations or truth tables are entered, it can generate the enhanced equations directly from the given equations, or can generate minimized equations using optimization routines. The optimizer generates optimized equations using the Quine-McCluskey's algorithm. If the description is given in equation form, the equations are transformed into a table form to permit the use of the same algorithm.

The code generator generates the simulation model from the model name, input pin names, output pin names, signal names, the optimized equations, and other information using domain rules. The code generation algorithm is based on a template matching mechanism (Biermann 1984).

5 MODEL VERIFICATION UNIT

When simulation models are automatically generated, it is necessary to verify the correctness of these models. Verification of models is accomplished in a similar fashion to the verification of design in that a simulator is used to verify new models. Some difficulty exists with respect to the determination of consistency among models at various levels.

Simulation at various levels is used in order to verify the correctness of models. The simulator can handle multi-level primitives according to the level of the primitives. The basic configuration is shown in Figure 8. From the given descriptions of models,

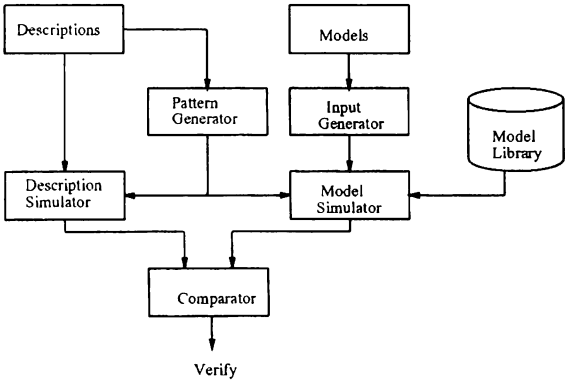

Figure 8: Model Verification Unit

expected behaviors are extracted and stored. This process is straightforward, to directly provide an environment for simulating a given model. If the user wants to see the results for each simulation pattern, they can be provided. If the user specifies the number of simulation patterns, then, according to the given number, the pattern set is generated. Also, if the user provides a simulation pattern set, it can be used for simulation. Then, according to the size of a described circuit, a random pattern set, or an exhaustive pattern set, may be generated by the pattern generator. Then, the simulation results are passed on to the comparator. After comparison, if all results are the same, the model is said to be correct. However, if any of results is not the same, an er-

ror message is provided and the simulation patterns in disagreement are provided to the user for further consideration.

6 MODEL LIBRARY AND KNOWLEDGE BASE

Model Library has several sub-libraries to store the same device using different levels of descriptions. The library handler is used to display all existing models and specifications. Also, it is used to place new specifications into the library. It stores and retrieves the circuit descriptions of various types and puts the generated models into the library. It is also used to link the structural models to their components and to add the generated elements to the specific simulators.

The Knowledge Base contains domain specific knowledge, used in model generation and verification. The knowledge stored in the Knowledge Base is classified into several categories including;

1) information about specific simulators, such as, logic values, memory state handling, how to call element routines, etc;

2) information about input descriptions, such as how to describe truth table specification, boolean equation specification, and HDLs; and

3) information about model verification.

7 POSTPROCESSING UNIT

The Postprocessing Unit generates an i/o interface for specific simulators. Since every simulator has its own way of using simulation models, the postprocessor should be capable of handling the variation. In order to do this, it requires various information about simulators, including; how to call an element routine, how to keep the state variables, etc. This information is stored in the Rule Base.

8 RESULTS

Using the AERO, many models were generated using ISCAS benchmark circuits (Brglez 1985). The results of model generation are shown in Table 1, including; circuit sizes, model generation times, number of simulation patterns, and simulation times. The circuit size is the number of gates, being assumed that the circuit was flattened into a gate level, if it is not at the gate level. Although model generation time increases according to the circuit size, the automatic model generation time is acceptable because of its speed and accuracy. For simulation purpose, 1024 random patterns were generated and used.

Simulation was performed using the generated models to prove thier efficiency and accuracy. The 3 value parallel fault simulation results are shown in Table 2. These results are derived using the PARSIM simulator (Kang 1990) For these results, 1024 random patterns are used.

Also, the mixed level path delay fault simulation results are shown in Table 3. For these results, 1000 random path delay faults and 1024 random patterns are used for benchmark circuits (Brglez 1989). In the table, the number of detected paths for robust

Circuit	# of Gates	Generation Time [sec]	# of Patterns	Simulation Time [sec]
c432	160	0.233	1024	7.013
c499	202	0.317	1024	14.317
c880	383	0.600	1024	19.600
c1355	546	0.833	1024	26.833
c1908	880	1.067	1024	35.067
c2670	1269	1.817	1024	47.817
c3540	1669	2.250	1024	64.250
c5315	2307	3.433	1024	106.433
c6288	2416	4.083	1024	163.083
c7552	3513	4.833	1024	261.833

Table 1: Results of Model Generation

Circuit	Faults	Fault Coverage[%]	CPU Time
c432	524	95.99	0.009 sec/pat
c499	758	97.49	0.032 sec/pat
c880	942	98.20	0.026 sec/pat
c1355	1574	89.07	0.183 sec/pat
c1908	1879	94.40	0.185 sec/pat
c2670	2747	81.61	0.580 sec/pat
c3540	3428	94.04	0.311 sec/pat
c5315	5350	95.73	0.534 sec/pat
c6288	7744	97.32	1.021 sec/pat
c7552	7550	91.08	1.345 sec/pat

Table 2: Results of Fault Simulation

Circuit	ROB	SNR	WNR	CPU[sec]
s713	556	144	242	51.14
s820	72	193	106	145.08
s832	131	185	131	152.03
s953	510	63	25	159.85
s1196	226	451	187	196.68
s1238	3	494	244	233.01
s1423	180	37	110	231.39
s1488	152	174	147	204.87
s1494	147	138	125	205.70
s5378	383	51	51	465.36

Table 3: Results of Delay Fault Simulation

test, strong non-robust test, and weak non-robust test, and simulation time for each circuit are represented.

As a final example, the design error simulation (Kang 1992) results are shown in Table 4. In the table, the number of design errors, simulation coverage metric, and simulation time for each circuit, are represented. For these results, 1024 random patterns are used.

Circuits	Errors	SCM [%]	CPU [sec]
c432	2489	97.99	2.15
c499	3012	98.24	3.60
c880	5171	98.36	6.73
c1355	7380	89.30	18.02
c1908	8775	94.21	10.43
c2670	13728	83.78	22.35
c3540	19130	94.88	38.75
c5315	33874	94.94	25.42
c6288	33472	92.46	33.67
c7552	38777	89.34	47.25

Table 4: Results of Design Error Simulation

These results show that automatic model generation is efficient, and the model generation time of the AERO is superior to that of experienced human programmers. Also, the results show that the generated models are efficient and accurate. If users are careful when writing the specifications, the generated code can approximate to the efficiency of the code written by a human programmer. Additionally, the generated models from the AERO can be used for various simulators.

9 CONCLUSION

The main purpose of this research was to find an easy, fast, and reliable way to generate simulation models and model libraries. To solve this difficult problem, a simulation model generation system using domain specific automatic programming techniques, is developed. The AERO can be used to construct simulation model libraries when developing new simulators or when upgrading existing simulators. The results prove that the time required to generate functional models is reduced considerably. In addition, since the designer needs not worry about the details of low level coding, the chance of errors in the design cycle can be significantly reduced. Therefore, this system could greatly reduce the cost of model generation for CAD systems and, consequently, could reduce the design cycle considerably.

REFERENCES

D. Barstow. 1985. Domain-Specific Automatic Programming. *IEEE Trans. on Software Eng.*

A. Biermann, G. Guiho and Y. Kodratoff. 1984. An Overview of Automatic Programming Construction Techniques. *Automatic Programming Construction Techniques.*

F. Brglez and H. Fujiwara. 1985. A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in FORTRAN. *Proc. of ISCAS*: 695-698.

F. Brglez, D. Bryan and K. Kozminski. 1989. Combinational Profiles of Sequential Benchmark Circuits. *Proc. of ISCAS*: 1929-1934.

C. Han, S. Kang and S. Szygenda. 1991. AFMG: Automatic Functional Model Generation System for Digital Logic Simulation. *Proc. of ASIC Conf.*

S. Kang, and C. Han. 1990. *PARSIM Manual.* Univ. of Texas, Austin.

S. Kang and S. Szygenda. 1992. Modeling and Simulation of Design Errors. *Proc, of ICCD*: 443-446.

S. Kang, B. Underwood and O. Law. 1994. Path-Delay Fault Simulation for a Standard Scan Design Methodology. *Proc. of ICCD.*

R. Spickelmier. 1989. *Release Notes for Oct Tools Distribution 3.0.* Univ. of California, Berkeley.

S. Szygenda. 1979. Simulation of Digital Systems: Where We Are and Where We May Be Headed. *Computer Aided Design*: 41-54.

H. Yang and S. Szygenda. 1991. A Domain Specific Automatic Programming System for Element Routine Generation. *Proc. of SSC.*

AUTHOR BIOGRAPHIES

JAE-WOOK LEE is a M.S. student in the Department of Electrical Engineering at Yonsei University, Korea. He received a B.S. degree in Electrical Engineering from Yonsei Univesity in 1996.

SUNGHO KANG is an Assistant Professor in the Department of Electrical Engineering at Yonsei University, Korea. He received a B.S. degree from Seoul National University, Korea and the M.S. and Ph.D. degrees in Electrical and Computer Engineering from the University of Texas at Austin, respectively. He was a senior staff engineer at the Semiconductor Systems Design Technology, Motorola Inc., a research scientist at Schlumberger Laboratory for Computer Science and a post doctoral fellow at the University of Texas at Austin. Dr. Kang's research interests include Simulation, Design Verification, VLSI CAD, Testing and Design for Testability.