Reprint from IASTED Int. Conf. on Applied Modeling, Simulation and Optimization (AMSO' 95), June]995, Cancun. Mexico

Automatic Code Generation for Simulators using Domain Specific Automatic Programming Techniques

Sungho Kang,
EE Dept.
Yonsei University
Seoul, Korea
e-mail: shkang@bubble.yonsei.ac.kr

Stephen A. Szygenda,* and Youngmin Hur
Chairman's Office, ECE Dept.
The University of Texas at Austin
Austin, Texas 78712
{szygenda@uts.cc, yhur@cerc}.utexas.edu

Abstract

The Automatic Simulator Generation System (ASG) is the first CAD tool which can automatically generate simulators and multi-valued simulation models, using domain specific automatic programming techniques. ASG can be used to develop new simulators, to upgrade existing simulators, and to generate multi-valued simulation models. This system represents a significant impact on simulation automation by providing the same environment for model generation and simulation.

Key words: CAD tool, Simulation, Simulator generation system, Domain specific programming.

1 Introduction

Automatic programming is concerned with methods for selecting programming constructs for: specification implementation, how to utilize fragmentary information, how to synthesize code from examples of the desired behavior, and how to utilize domain knowledge [1]. In domain specific automatic programming systems, which restrict the application domain [2], either the domain knowledge can be provided by the user as a part of the interactive specification process, or it may initially exist in the system. The methodology of representing domain knowledge, and the interaction between the domain knowledge and the programming knowledge, are dictated by the specific application domain. There have been some successful applications of domain specific automatic programming [3,4]. The concept of automatic code generation in the CAD area was introduced as an application of automatic functional model generation in 1979 [5]. However, until recently, functional elements have been developed exclusively through a knowledge intensive manual design process which is time consuming and error prone[6]. Recently, there have been several attempts to use domain specific programming in the generation of functional models [6]–[8].

*Designated Contact Person

Applying domain specific programming techniques to the design of simulators, has resulted in the Automatic Simulator Generation System (ASG). ASG is the first CAD tool which develops simulators automatically. Using this system, simulation automation can be achieved by providing the same environment for model generation and simulation. Also, in the case where a certain simulator is already being used, it can be used as an automatic model generator in order to upgrade the capabilities of the simulator.

Usually, a simulator is developed only once and used extensively. However, there are several cases where automatic simulator generation is important. Firstly, when designers design systems by using new design methodologies or by using new primitives which are not available in a library, many simulators cannot directly simulate these designs. In these cases a new simulator and/or new models are needed to verify these designs. Therefore, ASG can be used to develop special purpose simulators. This can be accomplished by users who are not familiar with the internal structure of simulators. Secondly, it can be used as an automatic multi-value simulation model generation system; since the generation of simulation models is the most difficult, error-prone and time-consuming process in the development or upgrading of simulators. Thirdly, A human programmer who has knowledge of simulation, can optimize an AMG simulator by modifying the code which was generated. The development time for this approach is much shorter than manual generation, with the same efficiency.

2 Automatic Simulator Generation System

The Automatic Simulator Generation System (ASG) is a tool which generates simulators automatically, using domain knowledge. There are 4 main steps that must be performed in this process. Specification is the process of acquiring all the necessary information about a target simulator. Design is the process of implementing the given specification. Refinement is the process of producing the code. Finally, verification is the process used to check the correctness of the generated programs. The global configuration of the ASG is shown in Fig. 1. This includes: User Interface, Preprocessor Generator, Model Generator, Evaluator Generator, and Merger.

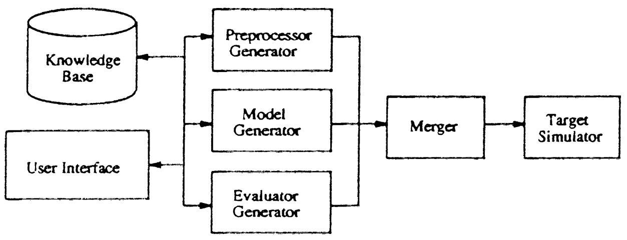

Figure 1: Automatic Simulator Generator

The User Interface is an interactive query system which allows users to easily describe the specification, to make decisions, and to override defaults that the system provides. It requires various types of information about the desired simulator. It has a simulator specification phase and a library specification phase. The simulator specification phase acquires the details of the target simulator, such as: input file descriptions, evaluation schemes, the number of logic values, etc. The library specification phase handles the basic elements in the library and updates new models. In order to update the new models, ASG handles the user's specification of models. Since there are many ways to describe the various models, the User Interface supports: schematics (OCT [9,10]), hardware description languages (SHDL [6], VHDL [11], and TDL [12]), netlists, boolean equations, and truth tables. When the user requests a new model, the system displays all available models and the user may use one of these library models. The block diagram of the User Interface subsystem is shown in Fig. 2.

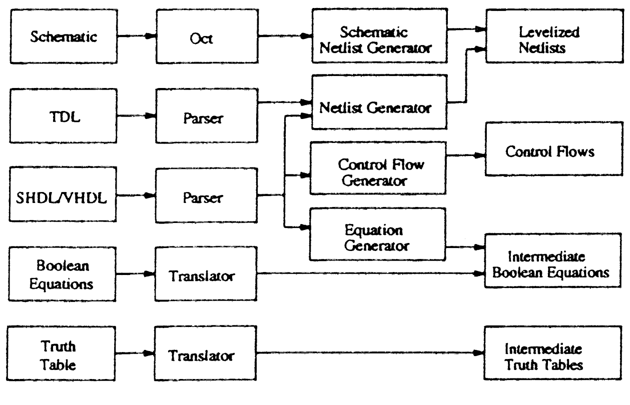

Figure 2: User Interface

From the schematic description, a net list is generated with the help of the OCT environment. A netlist can also be generated from the hardware description languages: Simulation Automation System (SAS) Hardware Description Language (SHDL), VHDL, and the Tegas Description Language (TDL). 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. When truth table descriptions or Boolean equations are entered, they are translated into internal data structures.

To implement efficient domain specific automatic programming techniques, the following must be considered. Firstly, knowledge of the application domain must be kept in some form of rule base to guide the program synthesis. Secondly, the user interface of a domain specific automatic programming system must provide the facilities to enable the users to input program specifications. Thirdly, due to the fact that the users' specifications are usually incomplete or imprecise, the system must be capable of figuring out the users' intent. The Knowledge Base includes basic simulation knowledge and provides intelligent programming for the generation of optimized code. The domain knowledge which is an integral part of the system, requires various details about simulators.

3 Preprocessor Generator

The Preprocessor Generator generates the parser and simulation vector translator for the simulator. For fault simulation, the fault list and fault collapsing code are generated. To describe the target designs, the input description format must be given. The system currently provides TDL [12], VHDL [11], and SHDL [6] formats. From the given options and input file names, the way to use the simulator is decided. Then, using the given formats of the input files, the parser and simulation pattern translator are generated. The routines to set up the internal data structures are also constructed. In ASG, data structures which are known to be efficient and general, are used as defaults. According to the user's specification, additional data structures may be constructed. The Element Table describes all elements used in the circuit, including primary inputs and primary outputs. The Signal Table includes all signals in the circuit. The table interactions are shown in Fig. 3.

The Element Table includes: element name, element id, corresponding output signal id, the number of fanins, fanin list, element type, etc. The element id is used as an index to identify a specific element. Since there are elements which have more than one output signal, an output signal id is required. The fanin list includes the signal ids of corresponding fanins. The Signal Table includes: signal name, signal id, corresponding element id, the number of fanouts, fanout list, storage for simulation values, etc. The fanout list includes the element ids of corresponding fanouts.

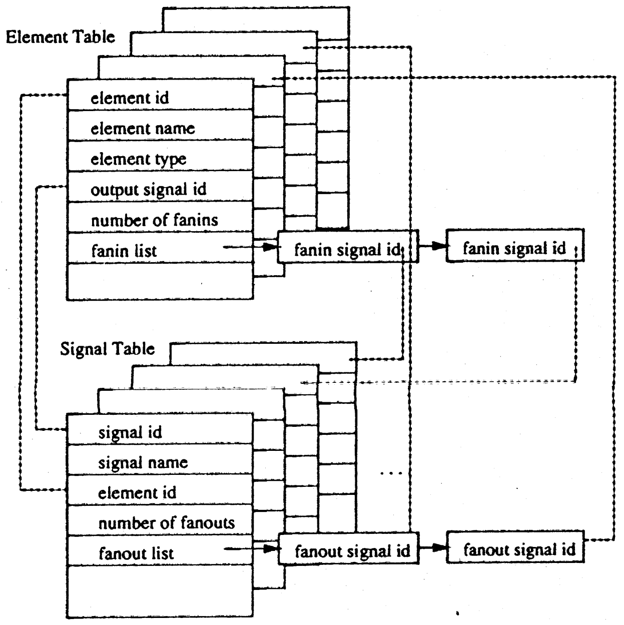

Figure 3: Table Data Structures

4 Model Generator

The Model Generator is the most important part of the ASG. It can generate various levels of simulation models, including hierarchical models. To generate functional models effectively, the following must be considered. Firstly, generated models should be able to easily interface to simulators. Secondly, generated models should be automatically verified. Thirdly, generated models should be efficient when executed in a simulator. Also, generated models should be concise in order to minimize the usage of memory. Finally, the generated models for sequential devices must have some mechanism to allocate data for each instance of the element, including internal memory states. The basic configuration of the Model Generator is shown in Fig. 4.

Using the information from the Preprocessor, the Model Generator synthesizes models written in the C language. Since most parallel simulators use two word representations, for three value simulation, two variables must be used to represent a variable defined in the original truth table or in the original boolean equations. According to the bit representations of a specific simulator, a signal modeler generates expanded table or an expanded equations. These expanded tables or expanded equations are passed to an optimizer. The optimizer generates optimized equations from the expanded table or expanded equations in order to reduce the size of the code.

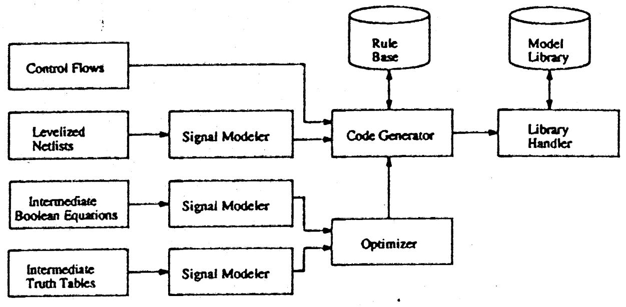

Figure 4: Block Diagram of the Model Generator

The code generator generates the simulation model using the optimized equations, module name, input names, output names, and signal names. The inputs of the code generator are the lists of control flows and the set of equations which are attached to the control flow. When the code generator generates the code, the two main steps are to model the primitives and to model the function, by synthesizing all the primitives. The code generation algorithm is shown in Fig. 5.

```
generate a header
declare the variables
if the model is a sequential one
    include state variables
initialize input parameters and state variables
for all control flows
    generate the code according to the control flow
    for all statements
        find and replace the primitive type
        include inputs and outputs
        generate the code
    connect to the other flows
generate the codes about the outputs of the model
```

Figure 5: Code Generation Algorithm

Initially, it generates a header which includes 'comments' and a 'model name'. Then, according to the number of inputs, number of state variables, and the number of primitives, the variables are declared. Also, temporary variables are declared. For sequential models, a memory is allocated to handle the internal states. Then, control flows are considered. The control flow types considered, are: 'while loop', 'do loop', 'for loop', and 'if statement'. This control flow includes a sequence of the statements (equations) which includes: inputs, outputs, their relationships and functionalities. The 'C' code is generated according to the functionality of each statement, the matched primitive, and the matched input and output variable names. These are continued until all control flows have been considered. Finally, the code about the interface of the outputs of the model is generated. This is done by setting the output parameters and saving the state variables.

For example, 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 (00), bit0 = 0 and bit1 = 1 for H (10), and bit0 = 1 and bit1 = 1 for X (11). The generated C model is shown in Figure 6.

```
/* name :  AND2 */
/* input :  a, b */
/* output :  c */
/* date :  March 1 1995 */

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 6: 2 input AND Gate Model

5 Evaluator Generator

The Evaluator Generator generates the evaluation routine which is the main routine for simulation. The domain knowledge required, includes: delay models, the number of logic values, hazard analysis, and simulation algorithms.

In zero delay simulation, no timing models are required. Nominal delay simulation allows the assignment of an average delay to each element type, and performs limited hazard and race analysis. Rise and fall delay simulation assigns different rising and falling times. Min-max delay simulation can specify ranges in which the elements can respond. The number of logic values used are; two values (0, 1), three values (0, 1, X), and five values (0, 1, U, D, E).

In fault simulation, the additional considerations are: inserting fault effects, propagating fault effects, and detecting fault effects. The fault simulation algorithms considered are; parallel fault simulation, concurrent simulation, and parallel pattern single fault propagation simulation.

For scheduling, the Evaluator Generator generates code related to a hybrid time queue, by default, since it is regarded as the most efficient. In this implementation, there is only one time wheel, and the macro time queue is implicitly implemented using a hashing method.

6 Merger

The Merger develops the simulator by compiling and linking all generated code. The generated code from the 4 generators, shown in Fig. 1, are transformed into better code using a C program beautifier. Using 'yacc' and 'lex', a circuit description parsers is generated. The code and circuit description parser are compiled. These are linked together with the model library, in an archive format, as shown in Figure 7.

If a user wants to upgrade the capability of a simulator, new models are required. This is done automatically by compiling the new models and adding them to an existing model library. Also, the names of the new models are updated into the file which contains the list of existing models. The updated file is compiled and linked together with an archive file containing all object files of the simulator.

7 Results

Using the Automatic Simulator Generation System, many simulators were developed. The results of simulator generation on a Sun4/110, are shown in Table 1, including characteristics of the simulators and simulator generation time. For example, it took only 9.534 sec to develop a 3 value zero delay simulator which has about 2000 lines of C. All simulators in the Table 1 use default models (from 2 to 8 input gates). These results show that automatic simulator generation is efficient and the simulator generation time is far superior to that of experienced human programmers.

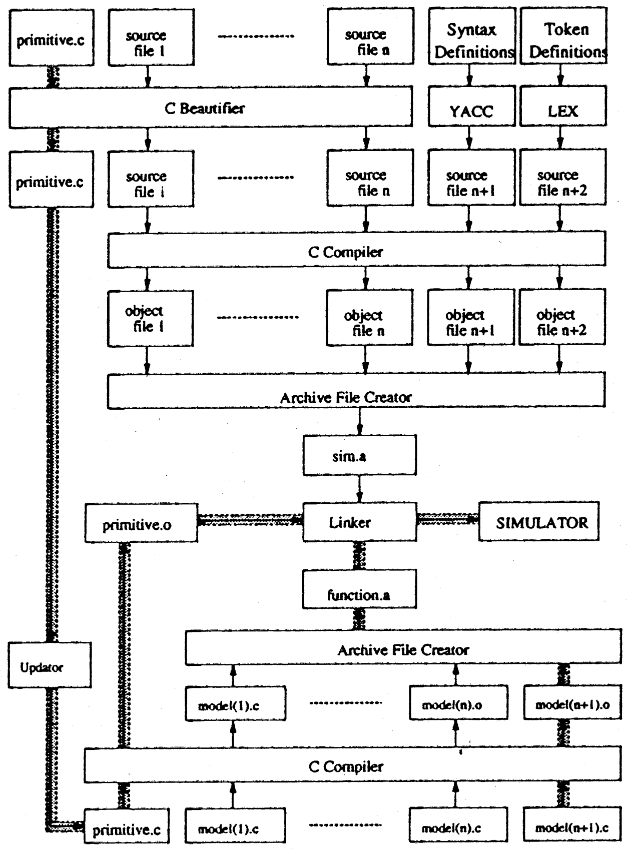

Figure 7: Merger

Table 1: Simulator Generation Results

Simulator	Functions	Simulator Generation Time
Logic	3 value, zero delay	9.534 sec
Logic	3 value, hazard analysis	13.142 sec
Fault	Parallel algorithm	11.712 sec
Fault	Concurrent algorithms	12.188 sec
Fault	PPSFP algorithm	11.274 sec

Using the automatically developed simulator, simulation was executed. The simulation results on a Sun4/110, are shown in Table 2, including: circuit size, logic simulation time, fault coverage, and fault simulation time. For these simulations, 1000 random patterns were used.

Table 2: Logic and Fault Simulation Results

Circuit	Sim. Time [sec/pattern]	Fault Coverage[%]	Fault Sim. Time [sec/pattern]
c432	0.008	95.99	0.009
c499	0.015	97.49	0.032
c880	0.022	98.20	0.026
c1355	0.027	89.07	0.183
c1908	0.035	94.40	0.185
c2670	0.105	81.61	0.580
c3540	0.065	94.04	0.311
c5315	0.106	95.73	0.534
c6288	0.163	97.32	1.021
c7552	0.263	91.08	1.345

The results show that the performance of the generated simulator is satisfactory. The automatically generated simulator is slower than one generated by an experienced human programmer, since the machine generated code is simple and the there is a lack of domain knowledge compared to the human knowledge. Also, the manually coded simulator achieved efficiency by including many time-saving heuristics. However, if more domain knowledge about simulation is provided to the Knowledge Base, more efficient simulators can be developed. Also, an experienced human programmer can upgrade the machine generated code to achieve more concise and efficient code. Upgrading is not difficult since the machine generated code is straightforward. This can be done much faster than by a human programmer developing a simulator from the beginning.

Using the Model Generator, many multi-valued element routines were generated after the generation of the simulators, in order to upgrade the performance. The results of model generation are shown in Table 3. Although, at a given level, model generation time increases according to the circuit size, the automatic model generation time is extremely fast when compared to manual generation of models.

8 Conclusion

An automatic simulator generation system was developed to provide an easy, fast, cost effective, and reliable way to generate simulators and functional models, using domain specific automatic programming techniques. This is the first attempt at automatic simulator generation and is another application of automatic programming. Future research includes the optimization of the generated code in order to generate more efficient programs, and the development of domain knowledge acquisition tools in order to more easily grow the domain knowledge.

Table 3: Model Generation Results

Circuit	Size [elements]	Generation Time [sec]
c432	160	0.68
c499	202	1.12
c880	383	3.35
c1355	546	5.82
c1908	880	13.68
c2670	1269	38.98
c3540	1669	47.33
c5315	2307	102.07
c6288	2416	97.67
c7552	3513	223.35
s208	158 (8 FFs)	0.28
s298	226 (14 FFs)	0.40
s344	235 (15 FFs)	0.68
s349	245 (15 FFs)	0.72
s386	275 (6 FFs)	0.61
s420	317 (16 FFs)	1.02
s444	328 (21 FFs)	0.83
s510	350 (6 FFs)	1.12
s641	567 (19 FFs)	3.38
s713	609 (19 FFs)	3.61
s820	582 (5 FFs)	1.93
s838	632 (32 FFs)	1.90
s953	658 (29 FFs)	3.68
s1196	822 (18 FFs)	5.58
s1238	824 (18 FFs)	5.23
s1423	950 (74 FFs)	9.55
s1488	946 (6 FFs)	7.83
s1494	944 (6 FFs)	7.75
s5378	4326 (179 FFs)	147.62
s9234	7226 (228 FFs)	545.71

9 References

[1] A. W. Biermann, G. Guiho and Y. Kodratoff, "An Overview of Automatic Program Construction Techniques," *Automatic Program Construction Techniques.* 1984.

[2] D. Barstow, "Domain-Specific Automatic Programming," in *IEEE Trans. on Software Engineering*, November 1985, pp. 1321–1336.

[3] E. Kant, "Synthesis of Mathmatical Modeling Software," in *IEEE Software* , May, 1993, pp. 30–41.

[4] D. Barstow, "Automatic Programming for Streams II: Transformational Implementation," in *Proceedings of the International Conference on Software Engineering*, 1988.

[5] S. A. Szygenda, "Simulation of Digital Systems: Where We Are And Where We May Be Headed," in *Computer Aided Design*, 1979, pp. 41–54.

[6] C. H. Han, S. Kang and S. Szygenda, "AFMG: Automatic Functional Model Generation System for Digital Logic Simulation," in *ASIC Conference*, 1991.

[7] C. Chuang and S. A. Szygenda, "The Automatic Element Routine Generator: An Automatic Programming Tool for Functional Simulator Design," *25th Annaul Simulation Symposium.* 1992.

[8] H. Yang and S. A. Szygenda, "A Domain-Specific Automatic Programming System for the Element Routine Generation," in *Proceedings of Summer Simulation Conference*, 1991.

[9] R. Spickelmier, *Release Notes for Oct Tools Distribution 3.0.* Electronics Research Lab., Univ. of California, Berkeley, 1989.

[10] D. Harrison, *Symbolic Editing with VEM.* Univ. of California, Berkeley, 1989.

[11] *IEEE Standard VHDL Language Reference Manual.* 1988.

[12] *TEGAS Language Reference Manual.* TEGAS Systems, Inc., GE Calma.