A Fault Simulator in Prolog: a Software Re-engineering Experience

J.H.M. Lee
Department of Computer Science
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong
Email: jlee@cs.cuhk.hk
Phone: +852 609 8426
Fax: +852 603 5024

Abstract

Prolog is employed in the re-implementation of a combinational circuit fault simulator. The input to the simulator is a combinational circuit written in a custom circuit description language. The parser of the simulator outputs an intermediate representation which is suitable for compilation, fault injection, and fault collapsing. The compilation phase amounts to program synthesis, which creates a Prolog program specifying the input-output relationship of the input circuit. The top level of the simulator is a meta-program, which guides the above activities, generation of input test vectors, simulation of the synthesized circuit program, and memory management. The resultant simulator, which was written in 31 pages of Prolog, executes at the same order of magnitude of speed as a similar version written in C. In addition, the Prolog simulator helps to reveal a bug in the C version of the simulator.

1 Introduction

There are two motivations for the project. First, a combinational circuit fault simulator **FAULTSIM** [7] was developed at the VLSI research group of the University of Victoria using the C language. Over the years, the simulator has become difficult to maintain and modify. Verification of the correctness of the simulator is infeasible as the implementation involves two inches thick of C code and documentation. This problem exists even when extensive software engineering technique was applied during the development. We propose to re-engineer[1] **FAULTSIM** using a declarative language since hardware description at the gate level is declarative in nature. In addition, declarative programs take shorter development time and are easier to maintain, modify, and verify.

Second, it is well-known [8, 3, 2, 4, 1, 6] that a subset of first-order predicate logic, namely Horn logic, is suitable for representing combinational circuit. For example, the behaviour of a two-input `NAND` gate and a `NOT` gate can be described by the following axioms:

```
nand( 0, 0, 1 ).      nand( 1, 0, 1 ).      not( 0, 1 ).
nand( 0, 1, 1 ).      nand( 1, 1, 0 ).      not( 1, 0 ).
```

We can define the behaviour of other logic gates similarly. Based on the gate axioms, the circuit in figure 1 can be defined by the following Horn clause:

```
circuit([X1,X2,X3],[X10]) :-
    fanout(X2,X4),fanout(X2,X5),not(X4,X6),not(X3,X7),
    nand(X1,X6,X8),nand(X5,X7,X9),nand(X8,X9,X10).
```

[1] By *re-engineering*, we mean the re-construction of a piece of software, the aim of which is to obtain a substantial improvement in the quality of the software.

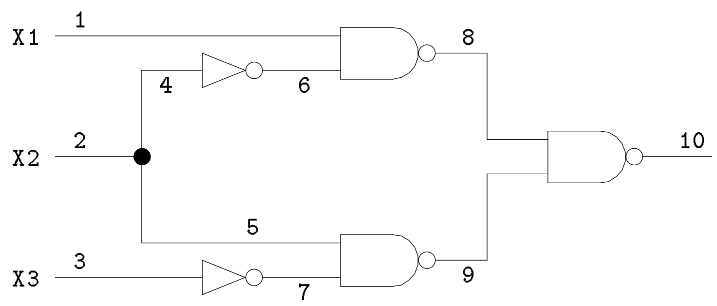

Figure 1: A simple circuit

The query "`?- circuit([0,1,1],[X10]).`" invokes the execution of the Prolog program with inputs "`{X1=0,X2=1,X3=1}`." The circuit becomes axioms in the Prolog program; simulation of the circuit becomes construction of a refutation out of the axioms and an initial query. The declarative nature of logic enables simulation to be performed in the backward direction. For example, the query "`?- circuit([X1,X2,X3],[1]).`" asks for the kind of inputs that would lead to an output of 1 in the circuit. More generally, circuit input and output variables can be left uninstantiated arbitrarily. *In view of the above reasons, we choose Prolog as the implementation language—Prolog is declarative and Prolog is based on Horn logic.*

FAULTSIM differs from other fault simulators in that its primary purpose is to serve as frontend to data compaction experiments. Outputs from simulations of the circuits are fed into data compactors of various models to generates signatures. Thus, we do not consider fault isolation nor test set generation but rather fault-free and stuck-at-fault simulations of the circuit. In addition, we support fault collapsing, which is an essential feature of a fault simulator. Fault collapsing is included in our project although it is not implemented in **FAULTSIM** yet.

We call our implementation **SIM**. A constraint on **SIM** is the compatibility to **FAULTSIM**, which accepts circuit description in a *custom* language. Since a library of circuit benchmarks has been collected over time, our simulator must be able to accept the **FAULTSIM** input syntax. **SIM** has four major components: the *input* module, the *fault injector*, the *compiler*, and the *fault collapsing* module. After the circuit description is read into the system, it is translated into an *intermediate representation* and stored in the system. Faults are injected into the intermediate representation before the representation is submitted to the compiler. The latter synthesizes a Prolog program that specifies the input-output relationship of the input circuit. The top level of **SIM** is a meta-program that co-ordinates the above activities, guides the execution of the synthesized programs, generates input test vectors, performs fault collapsing and memory management.

The rest of the paper is organized as follows. Section 2 describes the syntax of the circuit description language, and the scanner and parser. In section 3, we present the techniques used in the fault injection module. The compilation rules are discussed in details in section 4. Section 5 contains the fault collapsing rules and their corresponding clauses in Prolog. In section 6, we give a sample session of **SIM**. Section 7 summarizes the paper and sheds light on future work.

2 The Input Module

The input module consists of a *scanner* and a *parser*. We employ a simple lookahead scanner, which reads an input file and outputs a stream of tokens as Prolog atoms. A token is any number of consecutive characters not separated by white spaces. The token stream is passed to the parser to check the syntax and output a parse tree.

Before we discuss the parser, it is essential to know the grammar of the input circuit description language. Unfortunately, the grammar is not documented anywhere. By looking at several such circuit description files, we propose the following grammar in Definite Clause Grammar (DCG) rule form for the description language.

```
Circuit --> Header, Circuit_description.
Header --> Single_output_header | Multiple_output_header.
Single_output_header --> Output_line, Num_of_inputs.
Multiple_output_header --> 'M', Output_lines, Num_of_inputs.
Output_line --> 'positive integer'.
Output_lines --> List_of_lines.
Num_of_inputs --> 'positive integer'.
Circuit_description --> [] |
                        [Gate_description|Circuit_description].
Gate_description --> Output_line,Gate_type,Input_lines.
Gate_type --> 'AND' | 'NAND' | 'OR' | 'NOR' | 'XOR' | 'NXOR' |
              'NOT' | 'FAN'.
Input_lines --> List_of_lines.
List_of_lines --> [] | ['positive integer'|List_of_lines].
```

All strings are non-terminals except those in single quotes.

Once a grammar is defined, it becomes trivial to build a DCG top-down parser for the language. ALS Prolog [9], our implementation language, accepts the above grammar rules written in a slight variant of the syntax of DCG and translates them into a DCG parser. The parse tree generated by the parser, after some further processing, becomes the intermediate representation of the circuit in **SIM**.

The following is a circuit description of the simple circuit depicted in figure 1.

```
10
3
4 FAN 2 0
5 FAN 2 0
6 NOT 4 0
7 NOT 3 0
8 NAND 1 6 0
9 NAND 5 7 0
10 NAND 8 9 0
0
```

The parse tree generated is

```
circuit(header([10],3),[[4,fanout,2],[5,fanout,2],[6,not,4],
    [7,not,3],[8,nand,1,6],[9,nand,5,7],[10,nand,8,9]]).
```

In the header field of the parse tree, "`[10]`" is the list of output lines of the circuit (only line 10 in this case) and "3" is the number of input lines. The second field represents the structure of the circuit—how lines are connected to one another.

This is not the actual intermediate representation stored in the system. If we have a big circuit with many gates, say more than one hundred gates, then the parse tree will be huge. Unfortunately, ALS Prolog has a limit on the size of a clause. Consequently, the next step after parsing is to break the parse tree into, what we call, *chunks*, before it can be stored in the system. We also need to have some mechanisms to connect the chunks together. Suppose that we want to break the parse tree in the previous example in chunks of three gates. The result is

```
circuit([3,2,1],[10],[[4,fanout,2],[5,fanout,2],[6,not,4]]).
circuit([3,2,1,4,5,6],[10],[[7,not,3],[8,nand,1,6],[9,nand,5,7]]).
circuit([3,2,1,4,5,6,7,8,9],[10],[[10,nand,8,9]]).
```

The predicate `circuit` has three arguments. The first argument is the list of input lines to the chunk. The second one is the list of output lines of the chunk. The last argument remains to be a list of gate descriptions which describes the structure of the circuit (how gates are connected). The input lines to the next chunk carries all input lines to the current chunk plus the lines which are used as output lines of the gates in the current chunk. This is how information from one chunk is propagated to the next. For this to work properly, we need to add a slight restriction to the format of the circuit description language:

Not until a circuit line be used as output of a gate description can it be used as input lines.

In the current implementation of **SIM**, parse trees are broken down into chunks of one hundred gate descriptions.

3 The Fault Injector

SIM works with the stuck-at fault model [5]. Faults are of the form "line X stuck at Y." We write "X/Y," which means that circuit line X is stuck at the signal Y regardless of the input to the circuit. We only consider single stuck-at faults, i.e. we assume that there is only one fault present in the circuit.

When a line is faulty, the gate description with the line as output line becomes immaterial. The line is stuck at the certain value no matter what input is assigned to the gate. When a faulty line is used as input to a gate, we have to make sure that it is stuck at the faulty value instead of depending on output of another gate. Thus, fault injection involves two steps:

1. Removal of the gate description with the faulty line as output.

2. Injection of the stuck value into gate description with the faulty line as one of the input lines.

Since a gate description uses line number, we cannot just replace the number with the stuck value. We reserve four special line numbers for this purpose and this is express by the following axioms:

```
    % Stuck at +0.                          % Stuck at -0.
stuck_table( pos, 0, 901 ).            stuck_table( neg, 0, 903 ).
    % Stuck at +1.                          % Stuck at -1.
stuck_table( pos, 1, 902 ).            stuck_table( neg, 1, 904 ).
```

"Stuck at -X" is used to express negative line numbers [7].

Suppose the fault is line 5 stuck at 1. Continuing with our simple circuit example, the resultant intermediate representation is

```
circuit([3,2,1],[10],[[4,fanout,2],[6,not,4]]).
circuit([3,2,1,4,5,6],[10],[[7,not,3],[8,nand,1,6],[9,nand,902,7]]).
circuit([3,2,1,4,5,6,7,8,9],[10],[[10,nand,8,9]]).
```

The representation is fed to the compiler to generate a faulty circuit program for simulation.

4 The Compiler

We give a bottom-up description of the compiler. Every line in the circuit is compiled into a Prolog variable. Variables with the same name within a clause refers to the same individual. There are two ways to associate a circuit to a variable. First, we can maintain a pool of line-variable pairs, `(Line,LineVar)`. Variables are extracted from the pool. Lines with stuck-at-faults are represented by the pairs:

```
(901,0)          % stuck at 0
(902,1)          % stuck at 1
(903,1)          % -ve stuck at 0
(904,0)          % -ve stuck at 1
```

This is time-efficient especially when a variant of binary search is used to look up the pairs. However, the pool has to be carried around during the compilation phase even when only a few variables are needed (in case of small circuits). This takes up memory space. Second, variables can be generated on the fly. Generated variables are accumulated and looked up as required. This is analogous to the function of a symbol table. This is space-efficient but there is overhead in the lookup process. We have chosen the former method for time-efficiency.

A gate description is compiled into a series of Prolog goals by the following transformation rules:

1. The `gate` has only one input line.

   $$\texttt{[Out,gate,In]} \longmapsto \texttt{gate(InVar,OutVar)}$$

2. The `gate` has multiple *positive* input lines.

   (a) AND, OR, and XOR:

   $$\begin{array}{lcl} \texttt{[Out,gate,In}_1\texttt{,}\ldots\texttt{,In}_k\texttt{]} & \longmapsto & \texttt{gate(InVar}_1\texttt{,InVar}_2\texttt{,TmpVar}_1\texttt{),} \\ & & \texttt{gate(TmpVar}_1\texttt{,InVar}_3\texttt{,TmpVar}_2\texttt{),} \\ & & \vdots \\ & & \texttt{gate(TmpVar}_{k-2}\texttt{,InVar}_k\texttt{,OutVar)} \end{array}$$

   (b) NAND, NOR, and NXOR:

   $$\begin{array}{lcl} \texttt{[Out,ngate,In}_1\texttt{,}\ldots\texttt{,In}_k\texttt{]} & \longmapsto & \texttt{gate(InVar}_1\texttt{,InVar}_2\texttt{,TmpVar}_1\texttt{),} \\ & & \texttt{gate(TmpVar}_1\texttt{,InVar}_3\texttt{,TmpVar}_2\texttt{),} \\ & & \vdots \\ & & \texttt{ngate(TmpVar}_{k-2}\texttt{,InVar}_k\texttt{,OutVar)} \end{array}$$

3. The `gate` has negative input lines. An inverter is inserted into the circuit. For example,

   $$\begin{array}{lcl} \texttt{[Out,gate,-In}_1\texttt{,In}_2\texttt{]} & \longmapsto & \texttt{not(InVar}_1\texttt{,TmpVar),} \\ & & \texttt{gate(TmpVar,InVar}_2\texttt{,OutVar).} \end{array}$$

`LineVar` is the corresponding Prolog variable associated with `Line`.

Based on the above transformation rules, we compile a chunk of a circuit,

```
circuit(ListOfInputs,ListOfOutputs,ListOfGateDescriptions),
```

into a Prolog clause of the form

```
circuit(ListOfInputs,ListOfOutputs) :- CompiledGateDescriptions,
```

where `CompiledGateDescriptions` is the resultant Prolog goals obtained by compiling the gate descriptions. Clauses for each chunk will have slightly different names. Collecting all the compiled chunks (Prolog clauses), we get the Prolog program that can simulate the behaviour of the circuit. The compiled intermediate representation of our example is

```
circuit( [X3,X2,X1], [X10] ) :-
    fanout( X2, X4 ), not( X4, X6 ),
    c0circuit( [X3,X2,X1,X4,X5,X6], [X10] ).

c0circuit( [X3,X2,X1,X4,X5,X6], [X10] ) :-
    not( X3, X7 ), nand( X1, X6, X8 ), nand( 1, X7, X9 ),
    c1circuit( [X3,X2,X1,X4,X5,X6,X7,X8,X9], [X10] ).

c1circuit( [X3,X2,X1,X4,X5,X6,X7,X8,X9], [X10] ) :-
    nand( X8, X9, X10 ).
```

5 The Fault Collapsing Module

A circuit with n lines has $2n$ possible stuck-at faults, some of which exhibit the same faulty input-output behaviour. Suppose the input lines to an AND gate are numbered 1 and 2, and the output line is numbered 3. We cannot differentiate among the faults *line 1 stuck at 0*, *line 2 stuck at 0*, and *line 3 stuck at 0*. The time required for fault simulation can be reduced without loss of information by identifying equivalent faults and simulating only one fault from each equivalence class. Furthermore, some faults *dominate*[2] [5] others. We need not simulate the dominating faults. *Fault collapsing* is the process of identifying the equivalence fault classes and eliminating the dominating fault classes. We employ a two stage fault collapsing process. First, faults are grouped into equivalence classes by two-way implication (or fault equivalence) [5]. Second, dominating classes are removed by one-way implication [5]. One-way and two-way implications are summarized in the following axioms:

```
    % any_input/0 <==> output/0
    % any_input/1 ===> output/1
two_one_way( [Out,and|Ins], [[Out/0|Stuck_ins]], Out/1 ) :-
        stuck_at( Ins, 0, [], Stuck_ins ).
    % any_input/1 <==> output/1
    % any_input/0 ===> output/0
two_one_way( [Out,or|Ins], [[Out/1|Stuck_ins]], Out/0 ) :-
        stuck_at( Ins, 1, [], Stuck_ins ).
    % any_input/0 <==> output/1
    % any_input/1 ===> output/0
two_one_way( [Out,nand|Ins], [[Out/1|Stuck_ins]], Out/0 ) :-
        stuck_at( Ins, 0, [], Stuck_ins ).
    % any_input/1 <==> output/0
    % any_input/0 ===> output/1
two_one_way( [Out,nor|Ins], [[Out/0|Stuck_ins]], Out/1 ) :-
        stuck_at( Ins, 1, [], Stuck_ins ).
    % implication not applicable for XOR
two_one_way( [_,xor|_], [], none ).
    % implication not applicable for NXOR
two_one_way( [_,nxor|_], [], none ).
    % any_input/1 <==> output/0
    % any_input/0 <==> output/1
    % no one-way implication for NOT
two_one_way( [Out,not,In], [[In/0,Out/1],[In/1,Out/0]], none ).
    % implication not applicable for FANOUT
two_one_way( [_,fanout,_], [], none ).

%
% stuck_at( List, Value, Acc, List1 ) is true if List2 is a list
%     of pairs, "Ele/Value", obtained by attaching Value to each
%     element Ele in List (In case Ele is negative, the logical
%     complement of Value is used.) and List1 = List2 (+) Acc,
%     where (+) is the list append operator.
%
stuck_at( [], _, Acc, Acc ).

stuck_at( [H|T], Stuck_value, Acc, Stuck_ins ) :-
        H > 0,
        stuck_at( T, Stuck_value, [H/Stuck_value|Acc], Stuck_ins ).
```

[2] A fault α is said to *dominate* a fault β if every test that detects β also detects α.

```
stuck_at( [H|T], Stuck_value, Acc, Stuck_ins ) :-
        H < 0, not( Stuck_value, Stuck_value1 ),
        stuck_at( T, Stuck_value, [H/Stuck_value1|Acc], Stuck_ins ).
```

A fault collapsing exercise for our preceding example reveals the following fault classes: {5/1}, {2/0}, {2/1}, {1/1}, {9/1,5/0,3/1,7/0}, {8/1,1/0,4/1,6/0}, {3/0,7/1}, and {4/0,6/1}.

6 A Sample Session

In the following, we give a sample session of **SIM** simulating the SN74181 ALU/function generator circuit, which consists of 9 input lines, 6 output lines, 98 internal lines, and 30 logic gates (some of which have more than 2 input lines).

```
Welcome to SIM (TM) Version 3.0
Type "help" for help.

?- help.
"read_circuit(File).":  read File which contains a circuitdescription.
"simulate_fault_free.":  perform fault-free simulation.
"simulate_stuck_at_faults(Start,End).":  perform single stuck-at-fault
                                         simulation from line Start to End.
"perform_fault_collapsing.":  perform fault collapsing.
"simulate_fault_collapsing.":  perform fault collapsing and simulate the
                               circuit with a representative of each fault
                               class.
"halt.":  quit the system.

yes.
?- read_circuit( alu ).
Circuit Read and Parsed in 0.31666667 sec.

yes.
?- simulate_fault_free.
Fault Free Simulation Done in 3.4166667 sec.

yes.
?- simulate_stuck_at_faults( 1, 113 ).
Single Stuck At Faults Simulation Done in 780.98333 sec.

yes.
?- perform_fault_collapsing.
Fault Collapsing Done in 0.88333333 sec.

yes.
?- simulate_fault_collapsing.
Simulation with Fault Collapsing Done in 519.15 sec.

yes.
?- halt.
```

7 Concluding Remarks

SIM executes at the same order of magnitude of speed as **FAULTSIM**, which can simulate all stuck-at faults of SN74181 in around 750 seconds. **SIM** is implemented in approximately 1500 lines, including documentation, of Prolog code. Its code size compares favourably against that of **FAULTSIM**, especially when **FAULTSIM** does not support fault collapsing yet. **SIM** was developed in one month's time whereas **FAULTSIM** was developed for more than one year. The smaller code size and declarative nature of Prolog imply a more maintainable, modifiable and reliable system. Simulation output from **SIM** on the SN74181 circuit also helps to reveal a bug in the **FAULTSIM** implementation. Besides being used as a fault simulator, **SIM** can also be used for fast prototyping in circuit design. The designer only needs to encode the circuit in the circuit description language and **SIM** can provide a full simulation and analysis (by fault collapsing) of the circuit.

In the short run, an menu-type user-interface, one similar to that of **FAULTSIM**, should be built so that users who do not have experience with Prolog can adapt to **SIM** easily. There are several long term plans. First, code optimization at the parsing and compilation stages should be introduced. For this, we should be able to recognize the existence of sub-structures in the circuit description. Second, we should consider multiple stuck-at-faults and other fault models, such as bridging faults. Third, backward simulation, a distinct features made available by Prolog, should be investigated. Possible applications include fault isolation and test set generation. Fourth, we should consider applying logic programming to transistor level simulator with the encourage results in gate level simulation. Fifth, the circuit description language should be modified to incorporate the possibility of hierarchical design and circuit specification. Sixth, fault-free fanouts can be collapsed during the compilation phase as a compilation optimization.

Acknowledgement

The paper has benefited from discussions with Mantis Cheng, Patrick Lui, Micaela Serra, and Xiaoling Sun. Travel support is provided by the Hong Kong Telecom Foundation.

References

[1] W.F. Clocksin. Logic programming and digital analysis. *Journal of Logic Programming*, 4(1):59–82, March 1987.

[2] K. Eshghi. Application of meta-level programming to fault finding in logic circuit. In M. van Caneghem and D.H.D. Warren, editors, *Logic Programming and Its Applications*, pages 208–219. Ablex Pub. Corp., 1986.

[3] E. Gullichsen. Heuristic circuit simulation using PROLOG. *INTEGRATION, the VLSI Journal*, 3:283–318, May 1985.

[4] R. Gupta. Test-pattern generation for VLSI circuit in a Prolog environment. In E. Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, pages 528–535, London, United Kingdom, July 1986. Springer-Verlag.

[5] D.K. Pradhan. *Fault-Tolerant Computing—Theory and Techniques*, volume 1. Prentice Hall, 1986.

[6] P.B. Reintjes. A VLSI design environment in PROLOG. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium*, pages 70–81, Seattle, U.S.A., 1988. The MIT Press.

[7] X. Sun. The FAULTSIM manual. Unpublished manuscript, University of Victoria, 1988.

[8] D. Svanaes and E.J. Aas. Test generation through logic programming. *INTEGRATION, the VLSI Journal*, 2:49–67, 1984.

[9] C.M. White. *ALS Prolog User's Guide and Reference Manual—Sun Workstation Version 1.01*. Applied Logic Systems, Inc., August 1988.