A Manual of a C++ Implementation of PODEM

Contents

1 Introduction

PODEM [1] is an important ATPG algorithm which generate test patterns for stuck-at faults. In this manual, we will describe the internals of a reference implementation of PODEM. The contents include the usage, the data structure, algorithm implmentation and helping functions. After reading the manual, our intended audience should be able to perform gate-level netlist analysis and manipulation, modfiy the ATPG algorithm for their purposes, and extend it to other researches related to ATPG. Please note that this manual is not intended as an instruction of the algorithm. Familiarity of the algorithm and general concept of the C++ programming language is required.

2 Requirements

1. A Linux or Unix-equivelent system (including cygwin).

2. GCC 3.X.
3. make 3.8+
4. flex 1.875+
5. bison 2.5.4+
6. readline 5.0.4+ and ncurses 5.4.2+ library.

---

3 Installation

The installation is a simple one. At a Unix or Linux console, perform the following steps.

1. Enter the source code directory. For example, "cd podem".
2. Enter "make". An execution file "atpg" will be created.
3. (Optional) copy "atpg" to a directory searchable by your shell, e.g., cp atpg ~/bin.

To delete all generated object files and the executable, type "make clean".

---

4 Usage

The implementation is actaully a multi-purpose program. Depending on the version, it can perform several functions including logic simulation (single value or parallel pattern), fault simulation (stuck-at or transition), and ATPG (stuck-at or transition). The individual functions are selected with program command-line swtiches. They will be described in the following. Several examples will also be given at the next following section.

4.1 program interface

```
$ ./atpg -help
usage: atpg [options] input_circuit_file
        -help (print this help summary)
        -logicsim (run logic simulation)
        -plogicsim (run parallel logic simulation)
        -fsim (run stuck-at fault simulation)
      -stfsim (run single pattern single transition-fault simulation)
        -transition (run transition-fault ATPG)
        -input <$val> (set the input pattern file)
        -output <$val> (set the output pattern file)
        -bt [$val] (set the backtrack limit)
```

4.2 Examples

In the following, we assume that there is a benchmark circuit, s27.bench in the current directory. If applicable, the input test patterns are stored in s27.input. If we perform the ATPG, the output will be stored at s27.out.

Stuck-at fault ATPG

```
atpg -output s27.out s27.bench
```

Stuck-at fault ATPG with backtrack limit 1000

```
atpg -output s27.out -bt 1000 s27.bench
```

Transition fault ATPG

```
atpg -transition -output s27.out s27.bench
```

Logic simulation

```
atpg -logicsim -input s27.input s27.bench
```

Parallel logic simulation

```
atpg -plogicsim -input s27.input s27.bench
```

The -plogicsim function is identical to the -logicsim except that the program encode parallel patterns into one word to improve the performance.

Stuck-at fault simulation

```
atpg -fsim -input s27.input s27.bench
```

Transition fault simulation

```
atpg -stfsim -input s27.input s27.bench
```

Notice that all option swtiches can be abbreviated as long as they can be distinguishable. For example, we can use -l for -logicsim and -i for -input. Therefore, `atpg -l -i s27.input s27.bench` is the same as `atpg -logicsim -input s27.input s27.bench`.

4.3 Input Netlist File Format

The input netlist format for podem is ISCAS89 netlist format. In the format, each line denotes one gate including its output, function type and its fanins. For example, `10 = NAND(1, 3)` describe a two-input NAND gate. The output wire name is `10`, and the two inputs are `1` and `3`. It should be noted that the order of gates appearing in the netlist is not significant. The name of gates can be a string of the following alpha-numeric characters: 0-9, A-Z, a-z, _, [,].

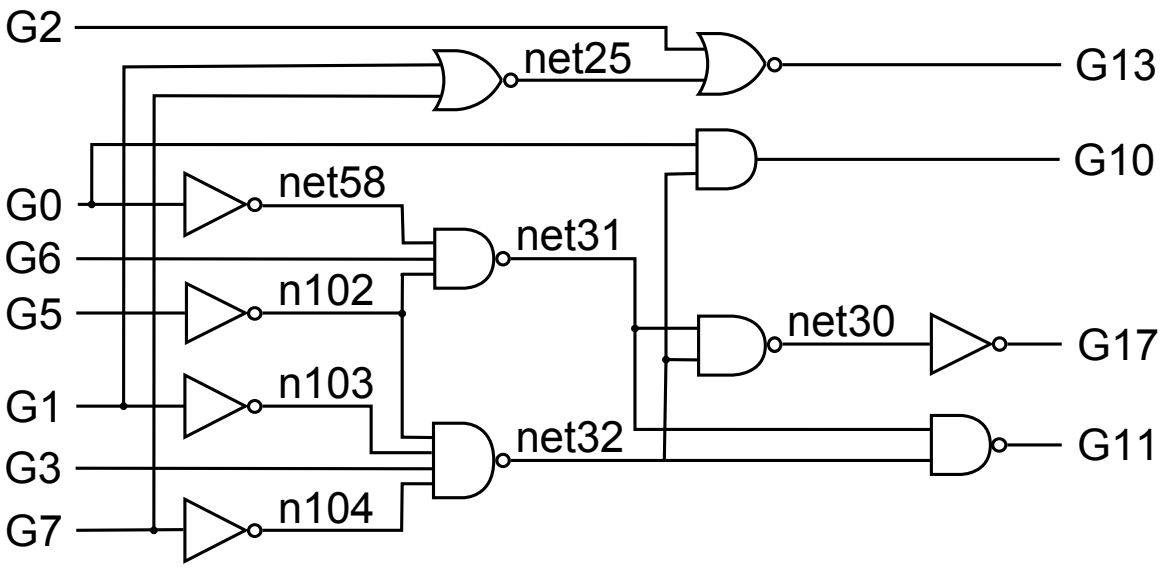
The first line of the file starts with # and is followed by the name of the circuit. The lines beginning with # excluding the first line are comment lines and ignored. These comment lines may be put into any part of the netlist.

Example netlists of the circuit s27 written in ISCAS89 netlist format is shown below.

```
# s27
INPUT(G0)
INPUT(G1)
```

```
INPUT(G2)
INPUT(G3)
INPUT(G5)
INPUT(G6)
INPUT(G7)
OUTPUT(G17)
OUTPUT(G10)
OUTPUT(G11)
OUTPUT(G13)
G17 = NOT(net30)
G10 = AND(net32, G0)
G11 = NAND(net32, net31)
G13 = NOR(G2, net25)
net32 = NAND(G3, n102, n103, n104)
net31 = NAND(net58, n102, G6)
net58 = NOT(G0)
n102 = NOT(G5)
n103 = NOT(G1)
n104 = NOT(G7)
net25 = NOR(G7, G1)
net30 = NAND(net31, net32)
```

And the schematic of the above s27 circuit is shown in the following figure:



The supported gate types are listed in the following table: Note that gate types can be also written in either upper or lower case.

Gate Type Identifier	Gate Functions
INPUT	primary input
OUTPUT	primary output
AND	and gate
NAND	nand gate
OR	or gate
NOR	nor gate
NOT	inverter
DFF	D-type Flip-flop

4.4 Input Test Pattern File Format

In the test pattern file, we use the first line to describe a list of primary inputs. They are identified with the keyword `PI`. In following lines, we print 0/1 bit information for each primary input in the order set by the above list. In the following, we have a list of test pattern, s27.input, for the above benchmark example s27.bench.

```
PI G0 PI G1 PI G2 PI G3 PI G5 PI G6 PI G7
0110010
1001001
1011000
1111000
0010111
0011100
1001100
0110000
1000010
```

5 Program Internals

5.1 File List

GetLongOpt.cc A public domain utility to support command line switches. For usage, please refer to `SteupOption()` in `main.cc`.

GetLongOpt.h Include file for `GetLongOpt.cc`.

ReadPattern.h Define `class PATTERN` to handle the file I/O of input test patterns.

atpg.cc Stuck-at fault ATPG.

circuit.cc Implement `class CIRCUIT` methods including `Levelize()`.

circuit.h Define `class CIRCUIT` to store the complete netlist.

fault.h Define `class FAULT` for single stuck-at fault. In this definition, if the fault is not on the branch, `Input` and `Output` will point to the same gate.

fsim.cc Implement single pattern parallel fault simulation for stuck-at faults.

gate.h Define `class GATE` to store all information of a gate. Note that we use two VALUE members: Value and Value_t to implement gate output values of two time frames.

hash.h A homemade hashtable utility.

main.cc Main program flow control.

psim.cc Implement parallel pattern logic simulation.

readcircuit.l Netlist character scanner for identifiers.

readcircuit.y Netlist parser.

sim.cc Implement single pattern logic simulation.

stfsim.cc Implement a single pattern single fault simulation for transition faults.

tfatpg.cc Transition fault ATPG.

tfault.h Define `class TFAULT` for transition faults.

typeemu.h Define various types and constants including `VALUE, GATEFUNC, FLAGS, ATPG_STATUS, FAULT_STATUS, CV, NCV and PatternNum`. Also define the two-input truth tables for all gates.

5.2 Important Functions

5.2.1 LogicSim()

The event-driven logic simulation basically check the `Queue[i]` each level `i` if there is any gate to be evaluated. If so, `Evaluate(gptr)` is used to calculated the output value based on fanins. If the computed output is different from the previous value, the computed value is set and we schedule all fanouts of the current gate (`gptr`). Note that to prevent multiple evaluation of a gate during one run of simulation, every scheduled gate will set the flag of `SCHEDULED` (it will be reset after evaluation).

5.2.2 GenerateAllFaultList()

Before fault simulation and ATPG, we will create the fault list by this function. No fault collapsing is performed here. Faults on different branches will set the `Branch` flag by `SetBranch(true)` can be identified by the `Is_Branch()`. All faults will be stored in `Flist`.

5.2.3 GenerateAllTFaultList()

This is the equivalent of the `GenerateAllFaultList()` for transition faults. All faults will be stored in `TFlist`.

5.2.4 Atpg()

The main flow-control function of ATPG process. From line 56 to 65 (in `atpg.cc`), it first create an output file stream for storing the generated test patterns if the "output" switch is set. It then generate patterns for each fault by calling `Podem()`. After the test generation, the function will print important fault statistics on the screen. Note that since we do not collapse faults, the "Equivalent FC" will be the same as "Fault Coverage".

5.2.5 Podem()

After reset all signal values to X, the following function is called to collect all gates at the fanout cone of the fault. The purpose is to enable the search of D-frontiers. (Therefore, there is no separate data structure to hold the current D-frontiers).

```
MarkPropagateTree(fptr->GetOutputGate());
```

Then the `SetUniqueImpliedValue()` is called to initialize the fault site (a 0 for stuck-at-1 fault) if any primary input can be determined uniquely without decisions. For example, for a stuck-at-1 fault at the output of a OR gate, and if the OR gate has two inputs connecting to primary inputs, we can set directly both primary inputs as logic 0. If any PI has implied value, we will perform logic simulation. And `FaultEvaluate(fptr)` is used to check if a fault is activated. If so, a faulty value D or B will be injected at the fault site. (Note that the faulty value will be inject the output gate of a branch fault, since we do not allow to store values at the branch only).

The main Podem process start with the following while loop:

```
while(backtrack_num < BackTrackLimit && status == FALSE) {
```

In the loop, we use `TestPossible(fptr)` to select a PI to either activate or propagate the fault. If such PI is decided, we will perform logic simulation and check again with `FaultEvaluate(fptr)` to prevent any masking of faults by newly-set PIs. Also we check by `CheckTest()` if any faulty effect is observed at POs. If so, the ATPG process is successful.

Since all decision PIs are stored at `GateStack`, at the event that we cannot select any new PI by `TestPossible(fptr)`, we will pop out the last decision PI from `GateStack`, and invert the last decision by `decision_gptr->InverseValue()`. If this has been done before (`decision_gptr->GetFlag(ALL_ASSIGNED)` is true), we will simply forget this decision, and continue to pop out the last PI from the `GateStack`. The above process is the backtracking algorithm used in Podem implementation.

References

[1] L.-T. Wang, C.-W. Wu, and X. Wen, *VLSI Test Principles and Architectures: Design for Testability*. Morgan Kauffman, 2006.