FAN ATPG

User Manual / Programmer Guide

Jan 20 2023

Lab of Dependable Systems
National Taiwan University

Copyright

Foreword

`FAN ATPG` is a package to generate patterns and run fault simulation.

The original authors of the package are **Lin Hio-Ting**, Hsu Po-Ya and Liao Kuan-Yu with LaDS, National Taiwan University.

Revision 2013
Han Cheng-Yu, Chen Ching-Yu, Chiang Kuan-Ying, Wang Ying-Hsu, Chen Po-Hao, Lin Kuo-Yu, Pan Chun-Han, Li Chia-An, Tsai Chia-Ling, Hsu Ling-Yun

Revision 2014
He Yu-Hao, Cai Zong-Yan, How Bo-Fan, Li Yu-Qing, Huang Guan-Yan, Xiao Zhi-Long, Lin Kun-Wei, Lin Shi-Yao, Chen You-Wei, Li Guan-De

Revision 2023
Wang Wei-Shen, Pan Yu-Hung, Chang Hsin-Tzu, Liang Zhe-Jia

The advisor of this open source project is Professor **James Chien-Mo Li**.

Conventions

The following conventions are used in this manual:

Italic

Used for terms or concepts when they are introduced for the first time.

`Consolas`

Used for showing source codes or pseudo code, as well as program elements within paragraphs such as statements, classes, macros, files and directories.

`Monospace Bold`

Used for commands or text that should be typed literally by the user.

`Monospace italic`

Used for commands or text that should be replaced with user-supplied values or by values determined by context.

`$` Represents the shell prompt.

1. Introduction

The LaDS *automatic test pattern generator* (ATPG) is developed for education purpose. This ATPG is based on the FAN algorithm. This is a comprehensive program that has almost all important functions that a typical ATPG tool should have. The source codes have good correspondence to the original FAN paper.

Users should read the chapters to understand how to use this tool. The programmers can further read the appendix to know how to modify the source codes.

We would like to thank all students who had contributed to this tool.

2. Commands and usage example

2.1. Commands

2.1.1. Command Summary

The following table contains a summary of the commands described in this manual. For the detailed descriptions and usages, please refer to the commands' reference page.

Command Summary

ATPG Commands	
Command Name	Description
add_fault	Add faults to the circuit.
add_pin_constraint	Add pin constraint on specified pin(s).
read_pattern	Read pattern from pattern file.
report_circuit	Report circuit information.
report_fault	Report fault information.
report_gate	Report gate information.
report_pattern	Report pattern information.
report_statistics	Report statistics on APTG/fault simulation result.
report_value	Report the gate's output value.
run_atpg	Run Automatic Test Pattern Generation.
run_fault_sim	Run fault simulation on the given pattern.
run_logic_sim	Perform logic simulation.
write_pattern	Write the pattern generated by ATPG to file.
MISC Commands	
Command Name	Description
report_memory_usage	Report total memory usage of resources.
report_pattern_format	Report pattern format of the program.
SETUP Commands	
Command Name	Description
build_circuit	Build netlist into internal circuit data structure.
read_lib	Read Mentor technology library file.
read_netlist	Read Verilog gate-level netlist.
report_cell	Report cell information.
report_lib	Report library information.
report_netlist	Report netlist information.
set_X-Fill	Set X-Fill to on/off.
set_dynamic_compression	Set Dynamic Test Compression to on/off.
set_fault_type	Set fault model type.
set_pattern_type	Set pattern type.
set_static_compression	Set Static Test Compression to on/off.

2.1.2. ATPG Command Descriptions

add_fault

Synopsis	add_fault [-h] [--help] [-f FILE] [--file FILE] [--all] [TYPE] [PIN]
Description	Add stuck at fault or transition delay fault to the circuit. You can choose which pins to insert the faults.
Arguments	FILE custom fault file TYPE Define the fault type. Can be SA0, SA1, STR, STF. PIN Pin location.
Options	-h,--help Print usage -a,--all add all faults

Example	Add all faults
	$ add_fault -a
	Add SA0 to pin G0
	$ add_fault SA0 G0

add_pin_constraint

Synopsis	add_pin_constraint [-h] [--help] PI... <0|1>
Description	You can limit the primary inputs to either 0 or 1.
Arguments	PI Primary inputs to be constrained. <0|1> Limit value to 0 or 1.
Options	-h,--help Print usage
Example	Limit pin G0 to 1 $ add_pin_constraint G0 <1>

read_pattern

Synopsis	read_pattern [-hv] [--help] [--verbose] FILE
Description	Read patterns from .pat file. We need to do it before running fault simulation. Then, we can run fault simulation based on the patterns read.
Arguments	FILE Pattern file
Options	-h,--help print usage -v,--verbose Verbose on. Default is off.
Example	Read test patterns from "s27_saf.pat" file. $ read_pattern pat/s27_saf.pat

report_circuit

Synopsis	`report circuit [-h] [--help]`
Description	Report circuit information. The name of netlist, number of PIs/PPIs/POs/PPOs/gates/nets will be printed on the screen.
Arguments	No argument needed.
Options	`-h,--help` print usage
Example	Report circuit information. `$ report circuit`

report_fault

Synopsis	`report_fault [-h] [--help] [-s STATE] [--state STATE]`
Description	Report fault information. Print out the fault model type in the beginning, as well as the number of faults. Then the fault information, including faults' type, and fault gates will be listed.
Arguments	No argument needed.

Options	`-h,--help`
	print usage
	`-s,--state STATE`
	only print out faults with given state STATE
	`UD` Undetected
	`PT` Possibly Testable
	`AU` Atpg Untestable
	`TI` Tied
	`RE` Redundant
	`AB` Atpg Abort
	currently support '`UD`', '`DT`', '`AU`', '`TI`', '`RE`', '`AB`' state

Example	Report fault information of faults with type UD (undetected). `$ report_fault –s UD`

report_gate

Synopsis	`report gate [-h] [--help] GATE1 GATE2 …`
Description	Print out the gate information, including gate ID, level, type, frame number, fanin number and fanout number, of the gates passed by argument.
Arguments	`GATE1, GATE2, …` Only the gate(s) with the given name(s) `GATE1,GATE2…` will be printed.
Options	`-h,--help` print usage
Example	Report gate information of U_G8, U_G9, and U_G10. `$ report gate U G8 U G9 U G10`

report_pattern

Synopsis	`report_pattern [-h] [--help] [--disable-order] PAT1 PAT2 …`
Description	Print out the information of input/output pattern. The order of PI/PPI/PO will be listed first if it is not disabled. In addition, the information of each pattern (generated by ATPG or read from pattern file) will be printed out, including its PI, PPI, PO, PPO values.
Arguments	`PAT1, PAT2, …` (Not implemented) Only the `PAT1`th, `PAT2`th, … pattern information of all the patterns will be reported.

Options	`-h,--help`
	print usage
	`--disable-order`
	Don't print out the PI/PPI/PO order information.

Example	Report current pattern information but do not show the order information. `$ report_pattern --disable-order`

report_statistics

Synopsis	`report_statistics [-h] [--help]`
Description	Show the statistics of ATPG/fault simulation results, which includes number of faults, fault coverage, number of patterns… etc. And you will also get the fault model, runtime information.
Arguments	No argument needed.
Options	`-h,--help` print usage
Example	Report statistics. `$ report_statistics`

report_value

Synopsis	`report_value [-h] [--help] GATE1 GATE2 …`
Description	All the specified gates' output value will be printed. The value printed is of the gates' fanout pin. And both the good value and faulty value will be printed.
Arguments	`GATE1,GATE2, …` Only the values of gate(s) with the given name(s) `GATE1,GATE2…` will be printed. If no name is specified, all gates' values will be reported.
Options	`-h,--help` print usage

Example	Print out values of all the gates.
	`$ report_value`
	Print out values of gate G0, G1, G2, and G3.
	`$ report_value G0 G1 G2 G3`

run_atpg

Synopsis	`run_atpg [-h] [--help]`
Description	Run ATPG based on FAN algorithm. It can generate a pattern to detect the fault(s) set before. In addition, it will also return 1. testable 2. untestable 3. abort for each fault during pattern generation.
Arguments	No argument needed.
Options	`-h,--help` print usage
Example	Run Automatic Test Pattern Generation. `$ run_atpg`

run_fault_sim

Synopsis	`run_fault_sim [-h] [--help] [-m METHOD] [--method METHOD]`
Description	Run fault simulation. It has two methods: parallel pattern and parallel fault. It will execute based on the given circuit, pattern, and fault type.
Arguments	No argument needed.

Options	`-h,--help`
	print usage
	`-m,--method METHOD`
	Simulation method.
	Choose either pp (parallel pattern) or pf (parallel fault) method.

Example	Run parallel pattern fault simulation.
	`$ run_fault_sim –m pp`
	Run parallel fault fault simulation.
	`$ run_fault_sim –m pf`

run_logic_sim

Synopsis	`run_logic_sim [-h] [--help]`
Description	Given a netlist and a set of input patterns, run logic simulation on the circuit and generate the output value of the circuit.
Arguments	No argument needed.
Options	`-h,--help` print usage
Example	Run logic simulation. `$ run_logic_sim`

write_pattern

Synopsis	`write_pattern [-h] [--help] [-f FORMAT] [--format FORMAT] FILE`
Description	Write the generated pattern to output file for further usage. After running the ATPG process, there will be a set of generated patterns. You can select a pattern format, and dump the pattern in the selected format to output file.
Arguments	`FILE`

	Defines the output pattern file name.

Options	`-h,--help`
	print usage
	`-f,--file FORMAT`
	Pattern format.
	Currently support ‘`pat`’, ‘`ASCII`’, ‘`STIL`’ and ‘`lht`’ format.

Example	Write pattern to “pat/s27_saf.pat” file in the format pat. `$ write_pattern –f pat pat/s27_saf.pat`

2.1.3. MISC Command Descriptions

`report_memory_usage`

Synopsis	`report_memory_usage [-h] [--help]`
Description	Print out total memory usage.
Arguments	No argument needed.
Options	`-h,--help` print usage
Example	Report memory usage. `$ report_memory_usage`

`report_pattern_format`

Synopsis	`report_pattern_format [-h] [--help]`
Description	Print out the input/output format of the pattern.
Arguments	No argument needed.
Options	`-h,--help` print usage
Example	Report pattern format. `$ report pattern format`

2.1.4. SETUP Command Descriptions

`build_circuit`

Synopsis	`build_circuit [-h] [--help] [-f NUM] [--frame NUM]`
Description	Store netlist information to circuit data structure. Users can determine how many time frames there are. If the number of time frames is not set by user or is set to lower than 1, the number of time frame is set to 1 by default.
Arguments	No argument needed.

Options	`-h,--help`
	print usage
	`-f,--frame NUM`
	number of frames

Example	Build circuit data structure based on the netlist read from Verilog file.
	`$ build_circuit`
	Set the number of frames to NUM and build circuit as in the above example.
	`$ build circuit –f NUM`

read_lib

Synopsis	`read_lib [-hv] [--help] [--verbose] lib_file`
Description	Read library file. It is necessary for both atpg and fault simulation to get manufacturing process information. Users can choose if they want to show the warning information in detail.
Arguments	`lib_file` mentor technology library file

Options	`-h,--help`
	print usage
	`-v,--verbose`
	Verbose on. Default is off.

Example	Read library information from "mod_nangate45.mdt" file.
	`$ read_lib techlib/mod_nangate45.mdt`
	Read library information from "mod_nangate45.mdt" file and show warning information of unconnected ports.
	`$ read_lib -v techlib/mod_nangate45.mdt`

read_netlist

Synopsis	`read_netlist [-hv] [--help] [--verbose] netlist_file`
Description	Read netlist information from Verilog file. It is necessary for both ATPG and fault simulation to get netlist information. Users can choose if they want to show the warning information in detail.
Arguments	`netlist_file` verilog gate level netlist file

Options	`-h,--help`
	print usage
	`-v,--verbose`
	Verbose on. Default is off.

Example	Read library information from "s27.v" file.
	`$ read_netlist netlist/s27.v`
	Read library information from "s27.v" file and show warning information of unconnected ports.
	`$ read_netlist -v netlist/s27.v`

report_cell

Synopsis	`report_cell [-h] [--help] [CELL]...`
Description	Report cell information in the netlist, including cell name and cell type.
Arguments	`CELL` If no name is specified, all cells will be reported.
Options	`-h,--help` Print usage
Example	Report cell information. `$ report_cell`

report_lib

Synopsis	`report_lib [-h] [--help]`

Description	Report library information, including the number of models, and all models' name.
Arguments	No argument needed.
Options	`-h,--help` Print usage
Example	Report library usage. `$ report_lib`

report_netlist

Synopsis	`report_netlist [-h] [--help] [--more]`
Description	Report netlist information, including number of modules, ports, cells, nets. It could also report the detailed information of modules, ports, cells and nets if –more.
Arguments	No argument needed.

Options	`-h,--help`
	Print usage
	`--more`
	Print more detailed information.

Example	Report netlist usage. `$ report_netlist`

set_X-Fill

Synopsis	`set_X-Fill [-h] [--help] on/off`
Description	This command is to do X-filling. We can choose if we want to do X-filling while running ATPG. The argument on/off is necessary, or this command will not be executed.
Arguments	`on/off` either on or off
Options	`-h,--help` Print usage

Example	Do X-filling during ATPG.
	`$ set_X-Fill on`
	Do not do X-filling during ATPG.
	`$ set_X-Fill off`

set_dynamic_compression

Synopsis	`set_dynamic_compression [-h] [--help] on/off`
Description	Set Dynamic Test Compression to on or off. We can choose if we want to do Dynamic Test Compression while running ATPG. The argument on/off is necessary, or this command will not be executed.
Arguments	`on/off` either on or off
Options	`-h,--help` Print usage
Example	Turn on dynamic compression mode. `$ set_dynamic_compression on`

`set_fault_type`

Synopsis	`set_fault_type [-h] [--help] fault_type`
Description	Set fault type. Currently supports stuck at fault and transition delay fault.
Arguments	`fault_type` Define the fault type. Can be '`saf`' or '`tdf`'.
Options	`-h,--help` Print usage
Example	Set fault type to stuck at fault. `$ set_fault_type saf`

`set_pattern_type`

Synopsis	`set_pattern_type [-h] [--help] pattern_type`
Description	Set pattern type. Currently support basic scan, launch on shift and launch on capture.
Arguments	`pattern_type` Define the pattern type. Could be BASIC, LOC or LOS.
Options	`-h,--help` Print usage
Example	Set pattern type to basic scan. `$ set pattern type BASIC`

`set_static_compression`

Synopsis	`set_static_compression [-h] [--help] on/off`
Description	Set Static Test Compression to on or off. We can choose if we want to do Static Test Compression during running ATPG. The argument on/off is necessary, or this command will not be executed.
Arguments	`on/off` either on or off
Options	`-h,--help` Print usage
Example	Turn on static compression mode. `$ set_static_compression on`

2.2. Usage example

After learning the commands in this tool, we will give some usage examples in this section with ATPG, Fault Simulation, and Logic Simulation.

You can simply run the execution file and type all the commands manually. An alternative is to create a script file and let the program automatically run all the commands in this file.

To do this, option `-f` is needed when executing the binary file:

```
$ ./ [execution file] -f [script file]
```

For example,

```
$ ./ bin/opt/fan –f script/atpg.script
```

2.2.1. Automatic Test Pattern Generation (ATPG)

Here is an example script to run SSF ATPG. First, we read the library file and netlist's information in order to build the circuit. Before ATPG, we set the parameters like fault model type, static compression mode and X-filling mode. After running ATPG, we write the generated patterns to a pattern file. Then, we report information including test coverage, fault coverage and atpg effectiveness, to a .rpt file. We can also write the pattern file for commercial tools. E.g. ascii format for Fastscan, stil format for Tetramax.

```
read_lib techlib/mod_nangate45.mdt
read_netlist netlist/s27.v
report_netlist
build_circuit --frame 1
report_circuit
set_fault_type saf
add_fault -a
set_static_compression on
set_X-Fill on
run_atpg
write_pattern  pat/s27_saf.pat
write_pattern  -f ascii  pat/s27_saf.ascii   // fastscan format
report_pattern
report_statistics > rpt/s27_saf_atpg.rpt
report_memory_usage
exit
```

atpg_ssf_s27.script

Following is another script to run transition delay fault ATPG for s27.

```
read_lib techlib/mod_nangate45.mdt
read_netlist netlist/s27.v
report_netlist
build_circuit --frame 2
report_circuit
set_fault_type tdf
add_fault --all
run_atpg
report_statistics > rpt/tdf/s27_tdfloc.rpt
write_pattern pat/tdf/s27_tdfloc.pat
write_pattern -f ascii pat/tdf/s27_tdf.ascii
exit
```

atpg_tdf_s27.script

Here are the script to run s27 transition delay fault ATPG in Launch-on-capture mode.

```
read_lib techlib/mod_nangate45.mdt
read_netlist netlist/s27.v
report_netlist
build_circuit --frame 2
report_circuit
add_scan_chains
set_fault_type tdf
add_fault --all
set_pattern_type LOC
run_atpg
report_statistics > rpt/tdf/s27_tdfloc.rpt
write_pattern pat/tdf/s27_tdfloc.pat
write_pattern -f ascii pat/tdf/s27_tdfloc.ascii
exit
```

atpg_tdfloc_s27.script

2.2.2. Fault Simulation

Here are some sample scripts to run Fault Simulation. Like ATPG, first we read the library and netlist information in order to build the circuit. Before Fault Simulation, we need to read the pattern and set fault with specific type and insert it into the circuit. After that, report information including test coverage, fault coverage and atpg effectiveness, to a .rpt file. By reading the report file, we can get information about simulation results.

```
read_lib techlib/mod_nangate45.mdt
read_netlist netlist/s27.v
report_netlist
build_circuit --frame 1
report_circuit
read_pattern pat/s27_saf.pat
report_pattern
set_fault_type saf
add_fault -a
run_fault_sim
report_statistics > rpt/s27_fsim.rpt
report_memory_usage
exit
```

fsim_ssf_s27.script

Here are sample scripts to run transition delay fault (loc) mode for s27.

```
read_lib techlib/modnangate_45.mdt
read_netlist netlist/s27.v
report_netlist
build_circuit --frame 2
report_circuit
add_scan_chains
set_fault_type tdf
add_fault --all
read_pattern pat/tdf/s27_tdfloc.pat
report_pattern
run_fault_sim
report_statistics > pat/tdf/s27_tdfloc.rpt
write_pattern -f ascii pat/tdf/s27_tdfloc.ascii
exit
```

fsim_tdfloc_s27.script

If you want to verify our TDFLOC patterns using Fastscan, you can do the following in Fastscan.

```
add_clocks 0 CK
add_scan_groups group1 s27.proc
add_scan_chain chain1 group1 test_si test_so
set_system_mode FAULT
set_pattern_source external  pat/tdf/s27_tdfloc.ascii
set_fault_type transition
add_faults -all
run
report_statistics
```

2.2.3. Logic Simulation

Here are some sample scripts to run Logic Simulation. First we read the library and netlist information so that we could build the circuit. Before Logic Simulation, we need to read the pattern and insert it into the circuit. After that, report the information of each gate value.

```
read_lib techlib/mod_nangate45.mdt
report_lib
read_netlist netlist/s27.v
report_netlist
build_circuit
report_circuit
read_pattern pat/s27_saf.pat
report_pattern
report_gate
run_logic_sim
report_value
exit
```

logicsim.script

3. Pattern format

3.1. Input/Output file format

In this section, we are going to introduce I/O pattern file format including ASCII format, STIL format and old version pattern format.

- **Input file format**
  If you want to do fault simulation with our tool, you need to give input patterns to the executable binary. The format of the input patterns must be the format decided by LaDS.

- **Output file format**
  There are three kinds of format to write patterns generated by this tool. The first one is the format decided by LaDS and can also be used to do Fault Simulation with this tool. The second one is ASCII format, which can be accessed by the commercial tool Fastscan. The third one is STIL format, which can be accessed by the commercial tool Tetramax.

3.1.1. Format of .pat file

```
Syntax

PI1 PI2 PI3 PI4 … |
FF1 FF2 FF3 … |
PO1 PO2 PO3 PO4 … |
[BASIC_SCAN|LAUNCH_ON_CAPTURE|LAUNCH_ON_SHIFT]
_num_of_pattern_#
_PATTERN_# pi1|pi2|ppi1|ppi2|po1|po2|ppo
```

The first three lines show the order of PIs, flip-flops in the scan chain, and POs. The fourth line specifies the pattern type. Basic Scan indicates the stuck at fault model. Launch On Capture and Launch On Shift indicate the transition delay fault model. The rest contents of the file specify the number of patterns and each pattern's information.

The following is an example of pattern set for stuck at fault model.

```
Example

G0 G1 G2 G3  |
U_G5 U_G6 U_G7  |
G7 G17
BASIC_SCAN
_num_of_pattern_2
_pattern_1 1101 |  | 010 |  | 01 |  | 101
_pattern_2 0111 |  | 011 |  | 10 |  | 010
```

3.1.2. Format of ASCII pattern

The ASCII file that describes the scan test pattern is divided into five sections, which is header_data, setup_data, functional_chain_test, scan_test, and scan_cell. We only introduce the details about setup_data, scan_test, and scan_cell in this section. The Header_Data section is an optional section which simply includes comments. Functional_chain_test is used to check whether a scan chain can work correctly. We don't focus on it here.

- Setup_data
  The definition of the scan structure and general test procedures that will be referenced in the description of the test patterns are contained in the setup_data section.

The data printed will be in the following format:

```
Syntax

SETUP =
   declare input bus "PI" = <ordered list of primary inputs>;
   declare output bus "PO" = <ordered list of primary outputs>;
   CLOCK "clock_name1" =
      OFF_STATE = <off_state_value>;
      PULSE_WIDTH = <pulse_width_value>;
   END;
   PROCEDURE TEST_SETUP "test_setup" =
      FORCE "primary_input_name1" <value> <time>;
      FORCE "primary_input_name2" <value> <time>;
      ....
      ....
   END;
   SCAN_GROUP "scan_group_name1" =
      SCAN_CHAIN "scan_chain_name1" =
         SCAN_IN = "scan_in_pin";
         SCAN_OUT = "scan_out_pin";
         LENGTH = <length_of_scan_chain>;
      END;
      PROCEDURE <procedure_type> "scan_group_procedure_name" =
         FORCE "primary_input_pin" <value> <time>;
         APPLY "scan_group_procedure_name" <#times> <time>;
      END;
    END;
END;
```

In the setup_data section, first we declare the list of primary inputs and primary outputs that are contained in the circuit. Then we define the list of clocks that are contained in the circuit. The clock data includes the clock name, the off-state value, and the pulse width value. *PROCEDURE TEST_SETUP "test_setup"* is an optional procedure that can be used to set nonscan memory elements to a constant state for both ATPG and the load/unload process. It is applied once at the beginning of the test pattern set. This procedure may only include force commands.

SCAN_GROUP "scan_group_name1" defines each scan chain group that is contained in the circuit. A scan chain group is a set of scan chains that are loaded and unloaded in parallel. The scan chain which is represented by *SCAN_CHAIN "scan_chain_name1"* defines the data associated with a scan chain in the circuit. If there are multiple scan chains within one scan group, each scan chain will have its own independent scan chain definition.

In the *SCAN_GROUP*, there are several procedures. The type of procedures may include shift procedure, load and unload procedure, shadow-control procedure, master-observe procedure, shadow-observe procedure, and skew-load procedure. *FORCE "primary_input_pin" <value> <time>* is used to force a value (0,1, X, or Z) on a selected primary input pin at a given time. The time values must not be lower than previous time values for that procedure. The time for each procedure begins again at time 0.

APPLY "scan_group_procedure_name" <#times> <time> indicates the selected procedure name is to be applied the selected number of times beginning at the selected time. This command may only be used inside the load and unload procedures.

- Scan_test

The scan_test section includes test patterns information. The following is the syntax of scan_test.

```
Syntax

SCAN_TEST =
  PATTERN = <number> [clock_sequential];
  FORCE “PI” “primary_input_values” <time>;
  APPLY “scan_group_load_name” <time> =
  CHAIN “scan_chain_name1” = “values....”;
  CHAIN “scan_chain_name2” = “values....”;
  ....
  ....
  END;
  FORCE “PI” “primary_input_values” <time>;
  MEASURE “PO” “primary_output_values” <time>;
  PULSE “capture_clock_name1” <time>;
  PULSE “capture_clock_name2” <time>;
  APPLY “scan_group_unload_name” <time> =
  CHAIN “scan_chain_name1” = “values....”;
  CHAIN “scan_chain_name2” = “values....”;
  ....
  ....
  END;
  ....
  ....
  ....
END;
```

To determine a pattern, you should use *PATTERN = <number>* to indicate pattern id first. *clock_sequential* means this pattern needs to concern the clock. Every pattern ends with *END;* to indicate the end of this pattern process. Using *FORCE* can assign values to primary inputs at a given time. *APPLY* is used for calling procedures like load, unload. You should also assign ppi/ppo values when calling load, unload procedure and give *END;* in the end of every *APPLY*. *MEASURE* is to measure the primary output values and compare with given values at given time. *PULSE* is to give a pulse to the assigned clock at a given time.

The following is the example of scan_test.

```
Example

SCAN_TEST =
pattern = 0 clock_sequential;
apply  "group1_load" 0 =
chain "chain1" = "010";
end;
force "PI" "0001101" 1;
measure "PO" "X1" 4;
pulse "/CK" 5;
apply  "group1_unload" 6 =
chain "chain1" = "101";
end;
end;
```

■ Scan_cell

The scan_cell section contains the definition of the scan cells used in the circuit. The scan cell data will be in the following format:

```
Syntax

SCAN_CELLS =
  SCAN_GROUP “group_name1” =
  SCAN_CHAIN “chain_name1” =
  SCAN_CELL = <cellid> <type> <sciinv> <scoinv>
  <relsciinv> <relscoinv> <instance_name>
  <model_name> <input_pin> <output_pin>;
  ....
  END;
  SCAN_CHAIN “chain_name2” =
  SCAN_CELL = <cellid> <type> <sciinv> <scoinv>
  <relsciinv> <relscoinv> <instance_name>
  <model_name> <input_pin> <output_pin>;
  ....
  END;
  ....
  END;
  ....
```

```
END;
```

cellid is a number that identifies the position of the scan cell in the scan chain. The number 0 indicates the scan cell closest to the scan-out pin.

type defines the type of flip-flops. The type may be MASTER, SLAVE, SHADOW, OBS_SHADOW, COPY, or EXTRA but we only use MASTER in our case here.

sciinv is T if there are odd inverters before the library input pin of the scan cell relative to the scan chain input pin. Otherwise, set F as its value.

scoinv is T if there are odd inverters before the library output pin of the scan cell relative to the scan chain output pin. Otherwise, set F as its value.

relsciinv is T if there are odd inverters relative to the library input pin of the scan cell. Otherwise, set F as its value.

relscoinv is T if there are odd inverters relative to the library output pin of the scan cell. Otherwise, set F as its value.

instance_name is the top level boundary instance name of the flip-flop in the scan cell.

model_name is the internal instance pathname of the flip-flop in the scan cell.

input_pin is the library input pin of the scan cell.

output_pin is the library output pin of the scan cell.

```
Example

SCAN_CELLS =

    scan_group "group1" =
        scan_chain "chain1" =
            scan_cell =    0  MASTER  FFFF  "/U_G7"  "I1"  "SI"  "Q";
            scan_cell =    1  MASTER  FFFF  "/U_G6"  "I1"  "SI"  "Q";
            scan_cell =    2  MASTER  FFFF  "/U_G5"  "I1"  "SI"  "Q";
        end;
    end;

end;
```

3.1.3. Format of STIL pattern

This format support the commercial tool Tetramax.

TODO: Finish this part of user guide with more complete information

4. Summary

This project provides basic ATPG training for our students in NTU. The codes are for educational purposes only. We thank to the contribution by all the authors to the source codes and this user guide. We hope you enjoy the training.

Appendix A How to add Commands, Arguments and Options

A.1 Adding Commands

We need to modify the following three files in the directory *atpg/pkg/fan/src* to create a new command.

```
main.cpp
atpg_cmd.h
atpg_cmd.cpp
```

Here, we will give an example to add the “hello” command which will print out “Hello World!”.

main.cpp

First, we need to create a *Cmd* object and use the function *regCmd* which is a member function in *cmdMgr* in the function **initCmd**.

```
Cmd *helloCmd = new HelloCmd("hello", &fanMgr);
cmdMgr.regCmd("ATPG", helloCmd);
```

atpg_cmd.h

Second, we need to define the class *HelloCmd* including constructor, destructor, and exec function which we will implement the functions this command has in atpg_cmd.cpp.

```
class HelloCmd : public CommonNs::Cmd {
  public:
      HelloCmd(const std::string &name, FanMgr *fanMgr);
      ~HelloCmd();

      bool exec(const std::vector<std::string> &argv);

  private:
      FanMgr *fanMgr_;
  };
```

atpg_cmd.cpp

Third, we implement the function which will print out “Hello world!”
Once we type “hello” command.

```
HelloCmd::HelloCmd(const std::string &name, FanMgr *fanMgr) :
            Cmd(name) {
            fanMgr_ = fanMgr;
          optMgr_.setName(name);
          optMgr_.setShortDes("hello");
          optMgr_.setDes("Print out Hello World!");
          Opt *opt = new Opt(Opt::BOOL, "print usage", "");
            opt->addFlag("h");
          opt->addFlag("help");
          optMgr_.regOpt(opt);
```

```
    }
HelloCmd::~HelloCmd() {}

bool HelloCmd::exec(const vector<string> &argv) {
    optMgr_.parse(argv);
    if (optMgr_.isFlagSet("h")) {
          optMgr_.usage();
          return true;
    }
          cout<<"Hello World!"<<endl;
    return true;
}
```

Here we use the member functions *setShortDes* and *setDes* in *OptMgr* and *addFlag* in *Opt* to help us understand what the hello command will do when we type `hello -h` or `hello --help` in the command line. After these steps, we now have a new command named "hello".

A.2 Example: Extend The Command "hello" with Arguments and Options

Here we will give a simple extended example based on the previous "hello" command example. It will have two additional features by using arguments and options.

```
$ hello [--name NAME]
$ hello file_name [--name NAME]
```

The first command will print out "Hello NAME!" and the second command will do the same thing while storing the text to the file named *file_name*. Here is the modified code:

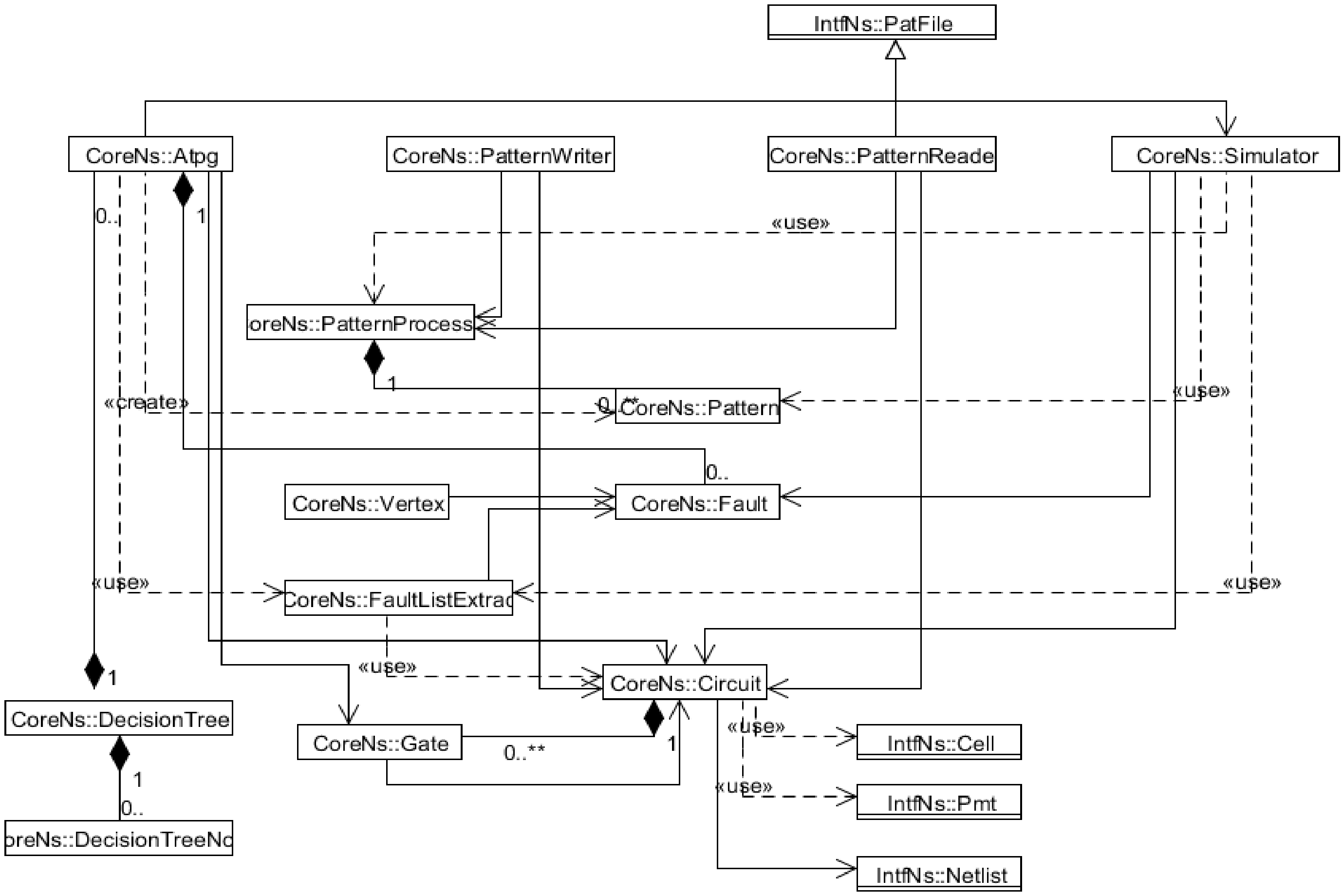
atpg_cmd.cpp

```
HelloCmd::HelloCmd(const std::string &name, FanMgr *fanMgr) :
Cmd(name) {
  fanMgr_ = fanMgr;
  optMgr_.setName(name);
  optMgr_.setShortDes("hello");
  optMgr_.setDes("Print out Hello World!");
  Opt *opt = new Opt(Opt::BOOL, "print usage", "");
  opt->addFlag("h");
  opt->addFlag("help");
  optMgr_.regOpt(opt);
        opt = new Opt(Opt::STR_REQ, "Name input", "NAME");
  opt->addFlag("name");
  optMgr_.regOpt(opt);
        Arg *arg = new Arg(Arg::OPT, "output file", "FILE");
  optMgr_.regArg(arg);
}
HelloCmd::~HelloCmd() {}

bool HelloCmd::exec(const vector<string> &argv) {
  optMgr_.parse(argv);
        if (optMgr_.getNParsedArg() == 1) {
        cout << "  Writing output to '" << optMgr_.getParsedArg(0) << "'
        ..."<<endl;
        ofstream output(optMgr_.getParsedArg(0).c_str());
        if(optMgr_.isFlagSet("name"))
              output<<"Hello "<<optMgr_.getFlagVar("name")<<"!"<<endl;
        else
              output<<"Hello World!"<<endl;

  }
  if (optMgr_.isFlagSet("h")) {
        optMgr_.usage();
        return true;
  }
        else if(optMgr_.isFlagSet("name")){
              string typeName = optMgr_.getFlagVar("name");
              cout<<"Hello "<<typeName<<"!"<<endl;
              return true;
        }
        cout<<"Hello World!"<<endl;
  return true;
}
```

Appendix B Data structure and UML

B.1 Data structure and Class Diagram

TODO: Modify this graph to function name with refactored version.



-private **+public** **#protected**

```
CoreNs::Atpg
Member data:
-pCircuit_ : Circuit *
-pSimulator : Simulator *
-currentTargetHeadLineFault_ : Fault
-numberOfHeadLine_ : int
-currentTargetFault_ : Fault
-headLineGateIDs_ : std::vector<int>
-gateID_to_n0_ : std::vector<int>
-gateID_to_n1_ : std::vector<int>
-gateID_to_valModified_ : std::vector<int>
-gateID_to_reachableByTargetFault : std::vector<int>
-gateID_to_lineType_ : std::vector<GATE_LINE_TYPE>
-gateID_to_xPathStatus_ : std::vector<XPATH_STATE>
-gateID_to_uniquePath_ : std::vector<std::vector<int>>
-circuitLevel_to_eventStack_ : std::vector<std::stack<int>>
```

```
-backtrackDecisionTree : DecisionTree
-backtrackImplicatedGateIDs_ : std::vector<int>
-gateIDsToResetAfterBackTrace_ : std::vector<int>
-initialObjectives_ : std::vector<int>
-currentObjectives_ : std::vector<int>
-fanoutObjectives_ : std::vector<int>
-headLineObjectives_ : std::vector<int>
-finalObjectives_ : std::vector<int>
-unjustifiedGateIDs_ : std::vector<int>
-dFrontiers_ : std::vector<int>
-isInEventStack : std::vector<int>
-firstTimeFrameHeadLine_ : Gate*
```

CoreNs::Circuit

```
Member data:
```

```
+pNetlist_ : IntfNs::Netlist *
+numPI_ : int
+numPPI_ : int
+numPO_ : int
+numComb_ : int
+numGate_ : int
+numNet_ : int
+circuitLvl_ : int
+numFrame_ : int
+tmeFrameConnectType_ : TIME_FRAME_CONNECT_TYPE
+totalGate_ : int
+totalLvl_ :int
+circuitGates_ : std::vector<Gate>
+cellIndexToGateIndex_ : std::vector<int>
+portIndexToGateIndex_ : std::vector<int>
```

```
Member function:
```

```
+buildCircuit(pNetlist:IntfNs::Netlist * const, numFrame :const
int & = 1, timeFrameConnectType: const TIME_FRAME_CONNECT_TYPE &
= CAPTURE) : bool
#mapNetlistToCircuit() : void
#calculateNumGate() : void
#calculateNumNet() : void
#createCircuitGates(): void
#createCircuitPI(): void
#createCircuitPPI() : void
#createCircuitComb(): void
#createCircuitPmt(gateID:const int &, cell:const IntfNs::Cell *
const, pmt:const IntfNs::Pmt * const): void
#determineGateType(gateID:const int &, cell:const IntfNs::Cell *
const, pmt: const IntfNs::Pmt * const): void
#createCIrcuitPO() : void
#createCircuitPPO() : void
#connectMultipleTimeFrame() : void
#assignFiMinLvl(): void
```

CoreNs:: DecisionTree

Member data:
-tree_ : std::vector<DecisionTreeNode>
Member function:
+clear() : void +put(gid:const int &, startPoint:const unsigned &) : void +get(gid:int &, startPoint:unsigned &) : bool +empty() :bool +lastNodeMark() : bool

CoreNs:: DecisionTreeNode

Member data:
+gid_:int +startPoint_: unsigned +mark_:bool
Member function:

CoreNs:: Fault

Member data:
+gateID_ : int +faultType_ : FAULT_TYPE +faultyLine_ : int +detection_ : int +faultState_ : FAULT_STATE +equivalent_ : int
Member function:

CoreNs:: FaultListExtract

Member data:
+gateIndexToFaultIndex_ : std::vector<int> +uncollapsedFaults_ : std::vector<Fault> +extractedFaults_ : std::vector<Fault> +faultsInCircuit_ : FaultPtrList +faultListType_ : FAULTLIST_TYPE
Member function:
+extractFaultFromCircuit(pCircuit:Circuit*) : void

CoreNs:: Gate

Member data:
+gateId_: int +cellId_: int +primitiveId_: int +numLevel_: int +frame_: int

```
+gateType_:Type
+numFI_:int
+faninVector_:std::vector<int>
+numFO_:int
+fanoutVector_:std::vector<int>
+atpgVal_:Value
+goodSimLow_: ParallelValue
+goodSimHigh_: ParallelValue
+faultSimLow_: ParallelValue
+faultSimHigh_: ParallelValue
+hasConstraint_: bool
+constraint_: ParallelValue
+cc0_: int
+cc1_: int
+co_: int
+depthFromPo_: int
+fiMinLvl_: int
+preValue_ : Value
---------------------------------
Member function:
---------------------------------
+isUnary(): Value
+isInverse(): Value
+getInputNonCtrlValue() : Value
+getInputCtrlValue() : Value
+getOutputCtrlValue(): Value
```

CoreNs:: Pattern

```
Member data:
---------------------------------
+primaryInputs1st_:std::vector<Value>
+primaryInputs2nd_:std::vector<Value>
+pseudoPrimaryInputs_:std::vector<Value>
+shiftIn_:std::vector<Value>
+primaryOutputs1st_:std::vector<Value>
+primaryOutputs2nd_:std::vector<Value>
+pseudoPrimaryOutputs_:std::vector<Value>
---------------------------------
Member function:
---------------------------------
+initForTransitionDelayFault(pCircuit : Circuit *)
```

CoreNs:: PatternProcessor

```
Member data:
---------------------------------
+staticCompression_ : State
+dynamicCompression_ : State
+XFill_ : State
type_ : Type
+numPI_ : int
+numPPI_ : int
+numSI_ : int
+numPO_ : int
+patternVector_ :std::vector<Pattern>
+pPIorder_ : std::vector<int>
```

+pPPIorder_ : std::vector<int> +pPOorder_ : std::vector<int>
Member function:
+init(pCircuit : Circuit *):void +StaticCompression() : void updateTable(mergeRecord : std::vector<bool>, patternTable : std::vector<bool>) : bool

CoreNs:: Vertex

Member data:
+data_ : Value * +fault_ : FaultList
Member function:

CoreNs:: PatternReader

Member data:
#curPattern_:int #pPatternProcessor_:PatternProcessor* #pCircuit_: Circuit*
Member function:
+setPiOrder(pPIs :const IntfNs::PatNames * const): void +setPpiOrder(pPPIs:const IntfNs::PatNames * const): void +setPoOrder(pPOs:const IntfNs::PatNames * const): void +setPatternType(patternType:const IntfNs::PatType &): void +setPatternNum(num:const int &): void +addPattern(pPI1:const char * const , pPI2:const char * const , pPPI:const char * const , pSI: const char * const , pPO1:const char * const , pPO2:const char * const , pPPO:const char * const):void #assignValue(valueVector :std::vector<Value>, pPattern :const char * const , size:const int &):void

CoreNs:: PatternWriter

Member data:
#pPatternProcessor_: PatternProcessor * #pCircuit_: Circuit *
Member function:
+writePattern(fname:const char * const):bool +writeLht(fname:const char * const):bool +writeAscii(fname:const char * const):bool +writeSTIL(fname:const char * const):bool

CoreNs:: Simulator

Member data:
-pCircuit_ : Circuit*

```
-numDetection_ :  int
-numRecover_ :  int
-events_ : std::vector<std::stack<int>>
-processed_ : std::vector<int>
-recoverGates_ : std::vector<int>
-faultInjectLow_ :
std::vector<std::array<ParallelValue, 5>>
-faultInjectHigh_ :
std::vector<std::array<ParallelValue, 5>>
-injectedFaults_[WORD_SIZE] : FaultListIter
-numInjectedFaults_ : int
-activated_ : ParallelValue
```

```
Member function:
```

```
+setNumDetection(numDetection:const int &) : inline void
+goodSim() : inline void
+goodSimCopyGoodToFault() : inline void
+goodValueEvaluation(gateID:const int &) : inline void
+faultValueEvaluation(gateID:const int &) : inline void
+assignPatternToCircuitInputs(pattern:const Pattern &) : inline
void
+eventFaultSim() : void
+parallelFaultFaultSimWithMultiplePattern(pPatternCollector:
PatternProcessor *, pFaultListExtract: FaultListExtract *) :
void
+parallelFaultFaultSimWithOnePattern(pattern: const Pattern &,
remainingFaults: FaultPtrList &) : void
+parallelFaultFaultSim(remainingFaults: FaultPtrList &) : void
+parallelPatternGoodSimWithAllPattern(pPatternCollector:PatternP
rocessor *) : void
+parallelPatternFaultSimWithPattern(pPatternCollector:
PatternProcessor *, pFaultListExtract: FaultListExtract *) :
void
+parallelPatternFaultSim(remainingFaults: FaultPtrList & ) :
void
-parallelFaultReset() : void
-parallelFaultCheckActivation(pfault: const Fault * const) :
bool
-parallelFaultFaultInjection(pfault: const Fault * const,
injectFaultIndex: const size_t &) : void
-parallelFaultCheckDetectionDropFaults(remainingFaults:
FaultPtrList &) : void
-parallelPatternReset() : void
-parallelPatternCheckActivation(pfault: const Fault * const ) :
bool
-parallelPatternFaultInjection(pfault: const Fault * const ) :
void
-parallelPatternCheckDetection(pfault: Fault * const ) : void
-parallelPatternSetPattern(pPatternProcessor: PatternProcessor
*, patternStartIndex: const int &) : void
```

B.2 Important Functions in ATPG and Fault Simulation

`CoreNS::ATPG`

This class includes all the functions that perform the FAN algorithm.
The following are some important functions in the class.
TODO: Modify this graph to function name with refactored version.

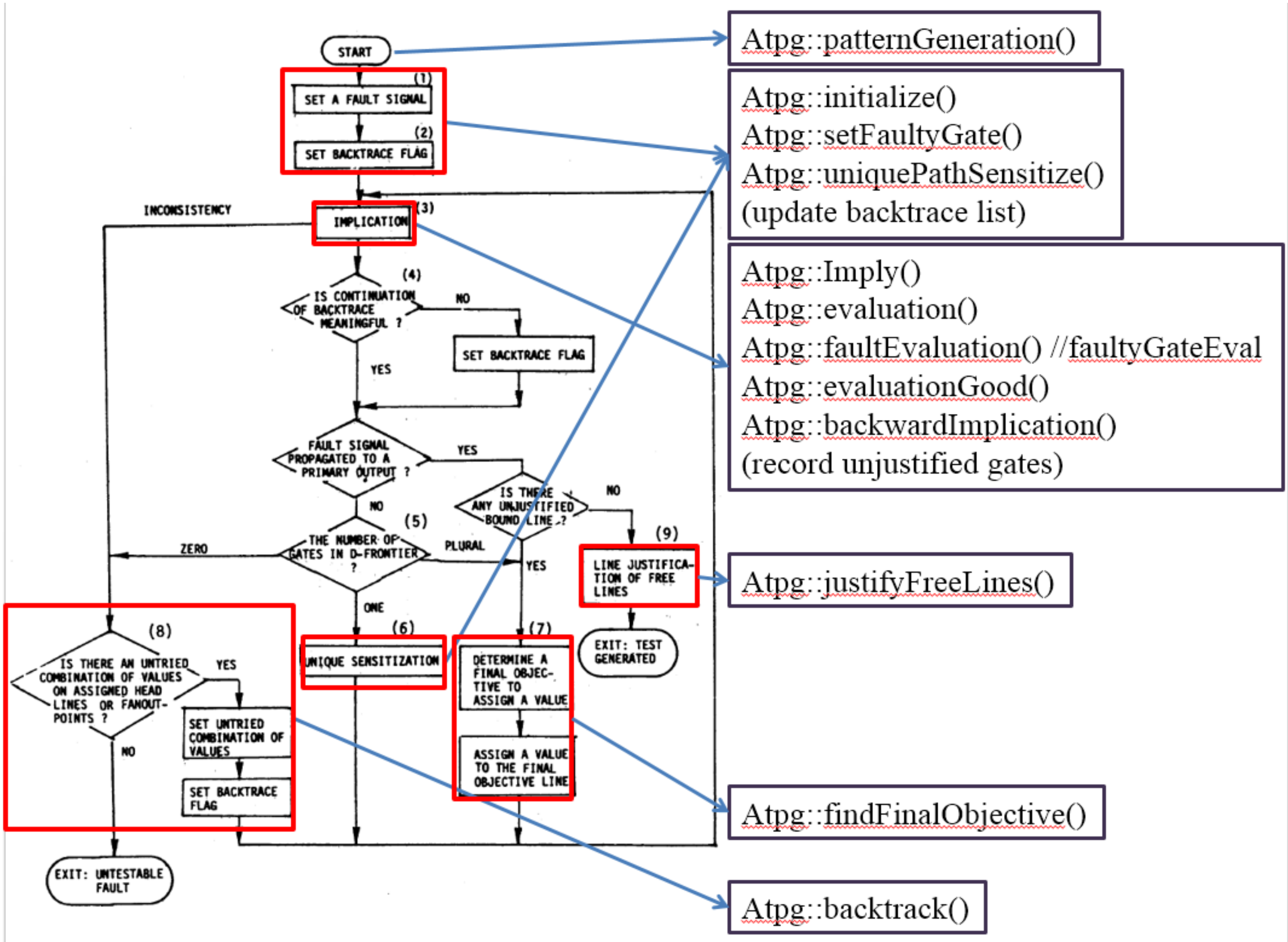


Figure above is the overall flow of FAN ATPG and the functions mapping to the algorithm.

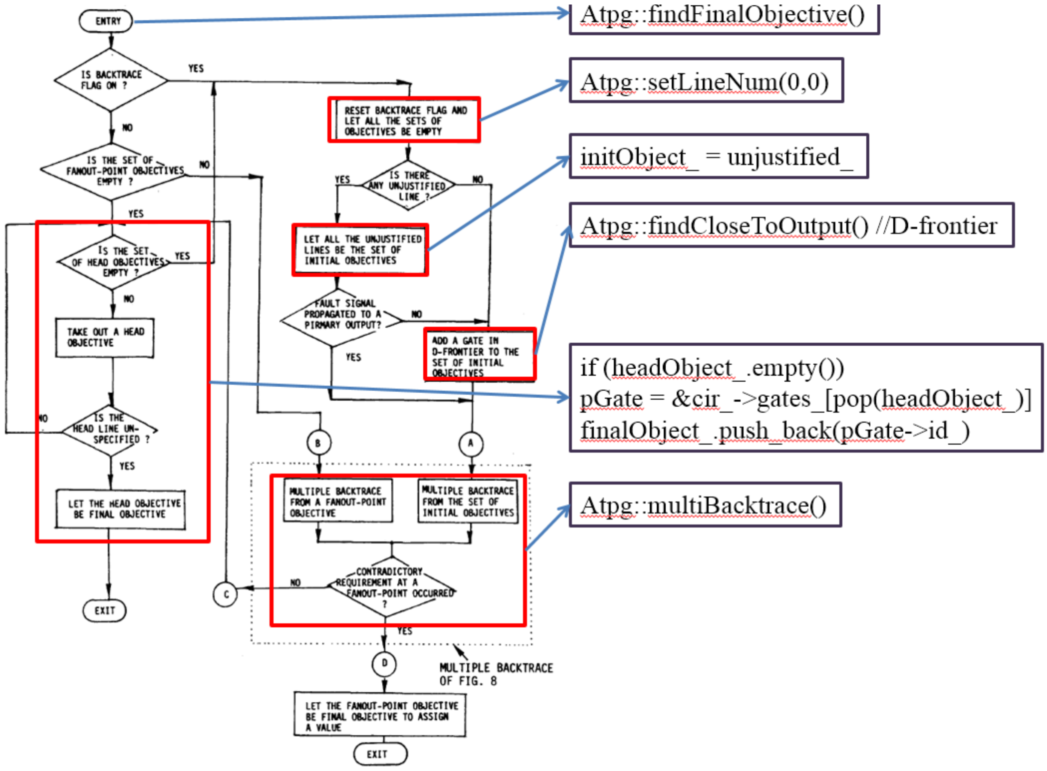


Figure above is find final objective and functions mapping to the algorithm.
TODO: Modify this graph to function name with refactored version.

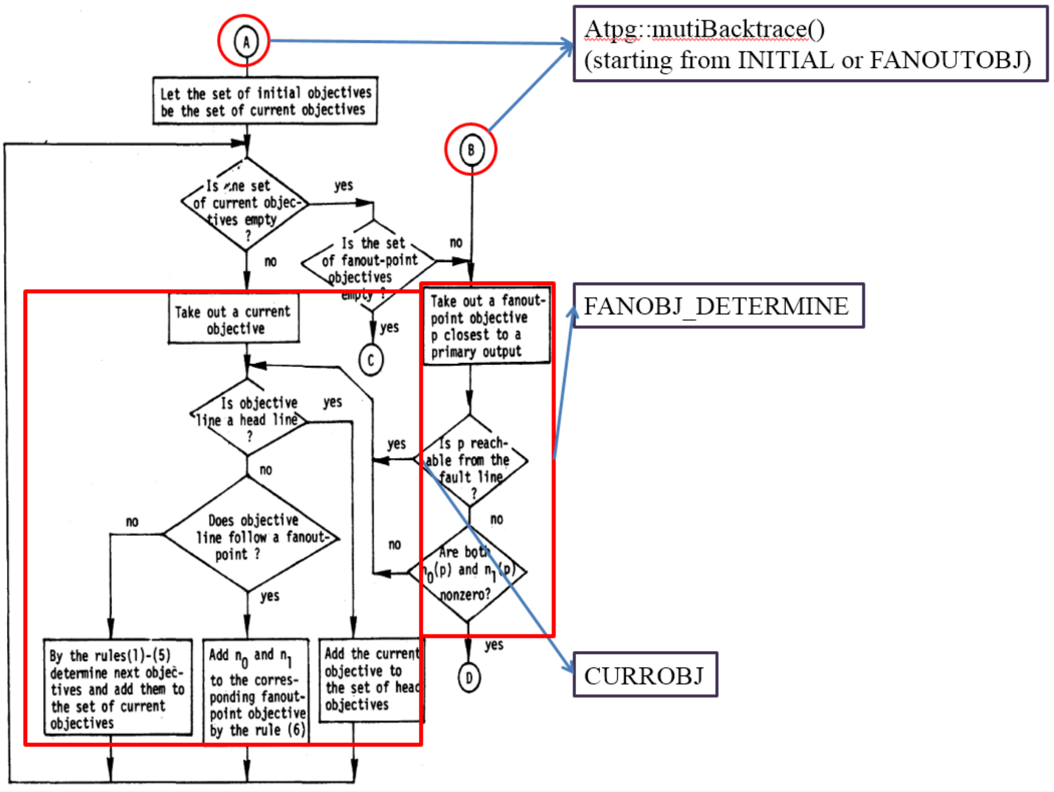


Figure above is multiple backtraces and functions mapping to the algorithm.
TODO: Modify this graph to function name with refactored version.

The flow diagram is mentioned in FAN ATPG original paper.
[On the Acceleration of Test Generation Algorithms]

The Atpg enums:

```
SINGLE_PATTERN_GENERATION_STATUS :
     PATTERN_FOUND
     FAULT_UNTESTABLE
     ABORT

GATE_LINE_TYPE :
     FREE_LINE
     HEAD_LINE
     BOUND_LINE

XPATH_STATE :
     NO_XPATH_EXIST
     XPATH_EXIST
     UNKNOWN

IMPLICATION_STATUS :
     FORWARD
     BACKWARD
     CONFLICT

BACKTRACE_STATUS :
     INITIAL
     CHECK_AND_SELECT
     CURRENT_OBJ_DETERMINE
     FAN_OBJ_DETERMINE

BACTRACE_RESULT :
     NO_CONTRADICTORY
     CONTRADICTORY
```

void Atpg::generatePatternSet (Pattern *pPatternProcessor,
FaultListExtract *pFaultListExtractor,
bool isMFO)

Synopsis	The main function of class Atpg
Description	This function generates a test pattern set based on an extracted fault list extracted from the target circuit. Activate STC/DTC depending on the pPatternProcessor's flag which is set previously in atpg_cmd.cpp based on user's script.
Arguments	[in, out] `pPatternProcessor` : A pointer to an empty pattern processor. It will contain the final test pattern set generated by ATPG after this function call. The test pattern set is generated based on the faults extracted from the target circuit. [in] `pFaultListExtractor` : A pointer to a fault list extractor containing the fault list extracted from the target circuit. [in] `isMFO` : A flag specifying whether the MFO mode is activated. MFO stands for multiple fault order, which is a heuristic with Multiple Fault Orderings.
Output	void

void Atpg::setupCircuitParameter()

Synopsis	Initialize the target circuit's parameters
Description	This function set up all the circuits' parameters and gates' parameters. Including circuitLevel_to_eventStack.
Arguments	void
Output	void

void Atpg::calculateGateDepthFromPO()

Synopsis	Calculate the depthFromPo_ of each gate
Description	This functions calculates the depth (how many gates) from PO/PPO of every gates. This function also initializes the this->gateID_to_valModified_ but should be moved to other places (TODO) for readability.
Arguments	A gate
Output	BackImpLevel

void Atpg::identifyGateLineType()

Synopsis	Identify and sets the gates' lineType
Description	This functions sets this->gateID_to_lineType_ (FREE_LINE, HEAD_LINE, BOUND_LINE). Records number of headline gates to this->numOfHeadLines_. Records all the headline gates' gateID into this->headLineGateIDs_.
Arguments	void
Output	void

void Atpg::identifyGateDominator()

Synopsis	Identify Dominator of every gate for unique sensitization.
Description	Traverse every gate and try to find each gates' Dominator. For each gate, if it has 1 or 0 fanout gate, we can skip it because a fanout free gate's Dominator is always the same. Push its fanout gates into circuitLevel_to_eventStack. We check the event stack for levels bigger than the gate level. In the process of finding the Dominator, we keep adding the fanout gates of gates into the event stack to traverse all paths the gate would pass.(For fanout of the gate, we just need to add its Dominator.). Once the event stack has only one gate left, we say that all paths will pass this gate, so this gate is the Dominator and we push this gate in this->gateID_to_uniquePath. We also check the existence of the Dominator in the process. The dominator doesn't exist when: 1. Event stack isn't empty but we find the PO/PPO in the event stack(numFO_ == 0). This implies more than one path to PO/PPO. 2. Event stack contains a fanout which has no dominator. Notice that the gateCount is equal to the number of events in the the whole event stack during this function call. If we have finished finding the Dominator of the gate, or the Dominator doesn't exist, gateCount will be 0 or set to 0. Then, we remove all the remaining events in event stack and go to next iteration(next gate). In addition, we check this->gateID_to_valModified_ to avoid repeated assignments.

	After this function, each gate has one or zero Dominator recorded in this->gateID_to_uniquePath_. A Dominator of a gate is the wire that must be passed for the dominated gate to reach PO/PPO.
Arguments	void
Output	void

void Atpg::identifyGateUniquePath()

Synopsis	Compute this->gateID_to_uniquePath_(2D vector). In unique path sensitization phase, we will need to know if the inputs of a gate is fault reachable. Then, we can prevent assigning non-controlling value to them. We find the Dominator, then we push_back the input gate which is fault reachable from the current gate. After identifying the unique path, if a gate has Dominator, this->gateID_to_uniquePath_ of this gate will contains the following gate id: [dominatorID fRIG1ID fRIG2ID ......] fRIG is faultReachableInputGate1ID for the above example Do NOT use fRIG in actual code for the sake of readability.
Description	We traverse all gates. For each gate, if it has no Dominator, we skip the gate. Now we push its fanout gates into the event stack. Notice that "count" is equal to the number of events in the whole event stack. We check the event stack for levels higher than the gate level. In this function, we keep adding the fanout of the current gate into the event stack to traverse all paths the gate would have to pass to reach PO/PPO. Simultaneously we adjust "count" and set the reachableByDominator of the fanout to current gate. Once "count" is 0 (the event stack has only one gate left), we should get the Dominator. Then we check reachableByDominator of the fanin of the Dominator. If it is the current gate, then we push the fanin into this->gateID_to_uniquePath. Finally, we go to the next iteration (the next gate).
Arguments	void
Output	void

**void Atpg::TransitionDelayFaultATPG(FaultPtrList &faultPtrListForGen,
PatternProcessor *pPatternProcessor,
int &numOfAtpgUntestableFaults)**

Synopsis	Do transition delay fault model ATPG
Description	This function is implemented very similar to Atpg::StuckAtFaultATPG() except for the following differences. 1. The fault model used is transition delay fault instead of stuck at fault. 2. Dynamic test compression is not implemented for transition delay fault.
Arguments	Please see the documentation of Atpg::StuckAtFaultATPG().
Output	void

**void Atpg::StuckAtFaultATPG(FaultPtrList &faultPtrListForGen,
PatternProcessor *pPatternProcessor,
int &numOfAtpgUntestableFaults)**

Synopsis	Do stuck at fault model ATPG on one fault and do DTC on the pattern generated to the single fault if the DTC flag is set to ON.
Description	The first fault pointed to by the first pointer in faultPtrListForGen will be selected as the first target fault for single pattern generation in this function. There will be three possible scenario after the single pattern generation on the first selected fault. 1. PATTERN_FOUND If a pattern is found for the first selected fault. A pattern will be allocated and push back topPatternProcessor->patternVector and will be updated immediately. If the DTC is set to ON for the pattern processor. The first selected fault will immediately be dropped by fault simulation. Then the DTC stage will start officially and select other undetected faults one by one for DTC. If the latter selected fault can be detected by filling some of the X(s) to 1 or 0. The fault state of the fault will be set to DT(detected) and the pattern will be updated to the original pattern with specific X(s) assigned. If the latter selected undetected fault is not detected, atpgVals will have to be restored to previous atpgVals because the single pattern generation will change the gate atpg values. If there is no more faults for DTC, the loop of DTC will be ended. After the loop, we will randomly XFill the pattern and perform fault simulation with the most recently updated pattern to drop the additional faults detected during the DTC phase. 2. FAULT_UNTESTABLE If the fault is not detected even after all the acktracks is done in the single pattern generation the fault is then declared as fault untestable. 3. ABORT If the Atpg is aborted due to the time of backtracks exceeding the BACKTRACK_LIMIT 500 (can be changed manually in namespace atpg.h::CoreNs).
Arguments	[in, out] `faultPtrListForGen`: Current list of fault pointers that are pointed to undetected faults. If detected when seen as the first selected target fault, it will be dropped immediately by fault simulation. If detected during DTC stage the faults will be dropped altogether after DTC. [in, out] `pPatternProcessor` : A pointer to pattern processor that contains a pattern vector recording the whole pattern set. In this function, the pattern processor should already possess the patterns generated for the faults before the current fault. A new Pattern will be pushed back to the the pPatternProcessor->patternVector_ if the fault first selected in this function is detected. It will become pPatternProcessor->patternVector_.back(). Then it will be determined and random XFilled at end of the function. [in, out] `numOfAtpgUntestableFaults` : It is a reference variable for recording the number of equivalent faults untestable. Here untestable faults means this function call has ended without abortion. If the function is aborted due to backtrack time exceeding limit, it is called aborted fault which is different to untestable fault.
Output	void

Gate *Atpg::getGateForFaultActivation(const Fault &faultToActivate)

Synopsis	Find and return the gate needed for fault activation.
Description	This function is used in DTC stage. Find and return the gate needed for fault activation.
Arguments	[in] `faultToActivate`: The latter fault selected to be activated in DTC stage.
Output	A gate pointer pointing to the gate needed to activate faultToActivate.

void Atpg::setGateAtpgValAndEventDrivenEvaluation(Gate &gate, const Value &val)

Synopsis	Directly set the output of "gate" to "value" and run evaluations by event driven.
Description	1. Call clearEventStack() and set gate.atpgVal_ to "val" 2. For each fanout gate of gate, push the gateID into the event stack if not in the event stack. 3. Do event driven evaluation to update all the gates in the event stack.
Arguments	[in, out] `gate` : The gate to set "val" to. [in] `val` : The "val" to assign to gate.atpgVal_.
Output	void

void Atpg::resetPrevAtpgValStoredToX()

Synopsis	Reset the `prevAtpgValStored_` of each gate to X.
Description	None
Arguments	void
Output	void

int Atpg::storeCurrentAtpgVal()

Synopsis	Store all the gates' atpgVal_ to prevAtpgValStored_ in the circuit.
Description	none
Arguments	void
Output	Count of values which changed from H/L to the value which is not the same as `prevAtpgValStored_`.

void Atpg::clearAllFaultEffectByEvaluation()

Synopsis	Clear all the fault effects before test generation for next target fault.
Description	none
Arguments	void
Output	void

void Atpg::clearFaultEffectOnGateAtpgVal(Gate &gate)

Synopsis	This function replace value of a gate from D/B to H/L.
Description	none
Arguments	[in] `gate` : The gate to have atpgVal_ cleared.
Output	void

Atpg::SINGLE_PATTERN_GENERATION_STATUS
Atpg::generateSinglePatternOnTargetFault(Fault targetFault, bool isAtStageDTC)

Synopsis	Given a target fault, generate a pattern.
Description	First, call initialize:

Set the pFaultyLineGate as the gate whose output is the target fault.
Then, set the backtraceFlag to INITIAL.
Keep calling doImplication() in a while loop, set the genStatus for latter return if PATTERN_FOUND, FAULT_UNTESTABLE, ABORT corresponding to the scenario of their literal meaning.

```
Loop content(while):
  IF number of backtracks exceeds BACKTRACK_LIMIT, ABORT
  IF doImplication() return false(conflicts):
    Clear the event stack and set this->gateID_to_valModified to all false.
    Call backtrack()
    If backtrack successful:
      Reset backtraceFlag to INITIAL for the latter findFinalObjectives(), set
      implicationStatus according to the BackImpLevel and reset
      pLastDFrontier to NULL
    Else IF backtrack failed meaning all backtracks have been finished but
    there is still no pattern found.
      => FAULT_UNTESTABLE
  Else IF doImplication() return true:
    IF continuationMeaningful() false:
      Then reset the backtraceFlag to INITIAL
    IF fault is propagated to any PO/PPO:
      IF there are any unjustified bound lines in circuit
        call findFinalObjective() and assignAtpgValToFinalObjectiveGates()
        and set implyStatus to FORWARD
      ELSE
        Justify all the free lines
        => PATTERN_FOUND
    ELSE:
      IF the number of d-frontiers is 0:
        backtrack()
        IF backtrack successful:
          reset backtraceFlag to INITIAL
          …
        ELSE:
          => FAULT_UNTESTABLE
      ELSE IF number of d-frontiers is 1:
        do unique sensitization:
          If  UNIQUE_PATH_SENSITIZATION_FAIL:
            continue back to the while loop(will backtrack, no more dFrontier)
          ELSE IF sensitization successful:
            implyStatus = BACKWARD and continue
          ELSE IF back implication level == 0
            continue
          ELSE IF nothing happened
            call findFinalObjective() and
            assignAtpgValToFinalObjectiveGates() and
            set implyStatus to FORWARD
```

There are four main atpgStatus for backtrace while generating the pattern:

IMPLY_AND_CHECK: Determine as many signal values aspossible then

	check if the backtrace is meaningful or not. DECISION: Using the multiple backtrace procedure to determine a final objective. BACKTRACK: If the values are incompatible during propagation or implications, backtracking is necessary. JUSTIFY_FREE: At the end of the process. Finding values on the primary inputs which justify all the values on the head lines.
Arguments	[in] `targetFault` : The target fault for this function to generate pattern on. [in] `isAtStageDTC` : The flag is true if this function is called during the DTC stage. See Atpg::initializeForSinglePatternGeneration() for more of how this flag affect the behavior of this function.
Output	SINGLE_PATTERN_GENERATION_STATUS, PATTERN_FOUND: Single pattern generation successful. A pattern is found for target fault. FAULT_UNTESTABLE: The target fault is not detected after all backtracks have ended. ABORT: The single pattern generation is aborted due to the time of backtracks exceeding the BACKTRACK_LIMIT(500).

Gate *
Atpg::initializeForSinglePatternGeneration(Fault &targetFault,
int &backwardImplicationLevel,
IMPLICATION_STATUS &implicationStatus,
const bool &isAtStageDTC)

Synopsis	This function replace value of a gate from D/B to H/L.

Description	First, assign fault to this->currentTargetFault_ for the future use of other functions.
	Then, assign the faulty gate to pFaultyLineGate. Initialize all the objectives and d-frontiers in Atpg.
	Initialize the circuit according to the faulty gate.
	IF gFaultyLine is free line,
	  Set the value according to Fault.type_.
	  SetFreeFaultyGate() to get the equivalent HEADLINE fault.
	  Assign this->currentFault_ to the new fault.
	  Set BackImpLevel to 0, implyStatus to FORWARD, faultyGateID to the new fault.gateID.
	ELSE
	  setFaultyGate() to assign the BackImpLevel and assign the value of fanin gates of pFaultyLineGate and itself. Add the faultyGateID to the this->dFrontier_.
	  Do unique sensitization to pre assign some values and then set implyStatus

	to BACKWARD. Last, If fault.type_ is STR or STF, setup time frames for transition delay faults.
Arguments	[in] `targetFault` : The target fault for single pattern generation, the faultyLine_ can be at input or output. [in, out] backwardImplicationLevel : The variable reference of backward implication level in single pattern generation, will be initialized according to the targetFault, and will be assigned to 0 if the implicationStatus is FORWARD. [in, out] `implicationStatus` : The variable reference of implication status in single pattern generation which indicates whether to do implication FORWARD or BACKWARD according to the targetFault. [in] `isAtStageDTC` : Specifying whether this function is called in the single pattern generation in DTC stage or not.
Output	The faulty gate corresponding to the fault.

void Atpg::initializeObjectivesAndFrontiers()

Synopsis	This function clear all the objectives, and most of the attributes of the circuit.
Description	none
Arguments	void
Output	void

void Atpg::initializeCircuitWithFaultyGate(const Gate &faultyGate, const bool &isAtStageDTC)

Synopsis	This function clear all the objectives, and most of the attributes of the circuit.
Description	Traverse through all the gates in the circuit. If the gate is free line, Set gateID_to_valModified_ to true. (free line doesn't need to be implicated/backtraced) Else Set gateID_to_valModified_ to false. Initialize this->gateID_to_reachableByTargetFault_ to all false. (All gate not reachable as default) Assign all gates' atpgVal_ to X if isAtStageDTC is false. (Keep the atpgVal_ from previous single pattern generation on first selected target fault or updated atpgVal_ during previous iteration in DTC) Initialize whole this->xPathStatus_ to UNKNOWN for future xPathTracing(). Set this->gateID_to_reachableByTargetFault_ to 1 and this->gateID_to_valModified_[gate.gateId_] to 0 for all the reachable fanout gate from the faultyGate.
Arguments	[in] `faultyGate`: The gate whose output is faulty. [in] `isAtStageDTC`: Specifies if the single pattern generation is at DTC stage.
Output	void

void Atpg::clearEventStack(bool isDebug)

Synopsis	Clear this->circuitLevel_to_eventStack_. Set this->gateID_to_valModified_, this->isInEventStack to 0.
Description	none
Arguments	[in] `isDebug`: Check this->isInEventStack_ correctness if the flag is true.
Output	void

bool Atpg::doImplication(IMPLICATION_STATUS atpgStatus, int implicationStartLevel)

Synopsis	Clear this->circuitLevel_to_eventStack_. Set this->gateID_to_valModified_, this->isInEventStack to 0.

Description	Enter a do while (backward) loop
	Loop content :
	  IF the status is backward:
	    Do evaluation backward starting from
	    this->circuitLevel_to_eventStack_[implicationStartLevel] to
	    this->circuitLevel_to_eventStack_[0].
	    Then, do evaluation() forward from
	    this->circuitLevel_to_eventStack_[0..totalLevel]
	evaluateAndSetGateAtpgVal() will return
	  FORWARD : do nothing
	  BACKWARD :
	    Do nothing if doing evaluations backward.
	    If doing evaluations forward, immediately break current loop and go back to the loop doing backward evaluations in the event stack.
	  CONFLICT : any failed evaluations

Arguments	[in] `atpgStatus`: Indicating the current atpg implication direction (FORWARD or BACKWARD) [in] `implicationStartLevel`: The starting circuit level to do implications in this function.
Output	A boolean, Return false if conflict after evaluateAndSetGateAtpgVal() Return true if no conflicts for all implications

Atpg::IMPLICATION_STATUS Atpg::doOneGateBackwardImplication(Gate *pGate)

Synopsis	Do backward implication on one gate.
Description	This function is specific designed for evaluateAndSetGateAtpgVal() to call when pGate's atpgVal_ can’t be evaluated due to the lack of determined gate inputs’ values. This function is aimed to keep doing implication backward starting from pGate. It will return FORWARD when reach PI/PPI or is unable to justify atpgVal_, otherwise it will return BACKWARD. Note that this function will never return CONFLICT.
Arguments	[in] `isDebug`: Check this->isInEventStack_ correctness if the flag is true.
Output	IMPLICATION_STATUS, Whether to implicate forward or backward

bool Atpg::backtrack(int &backwardImplicationLevel)

Synopsis	When we backtrack a single gate in the decision tree, we need to recover all the associated implications starting from the startPoint of its DecisionTreeNode as a index in this->backtrackImplicatedGateIDs_.
Description	Check if the decisionTree_.get(...) is true If true : DecisionTreeNode already marked, Pop it from decision tree and check next bottom node. Else : Update the unjustified lines. Backtrack the gate.atpgVal_ from previous decisionTree_.get(), reset all the gate in this->backtrackImplicatedGateIDs_ to not modified and the value to unknown. Recalculate the backward implication level, reconstruct the event stack, this->dFrontiers_, this->unjustifiedGateIDs_, this->xPathStatus. return true, indicating backtrack successful. If no more gate(node) in decision tree to backtrack than return false, indicating backtrack failed, fault untestable.

```
The decision gates are put in the decisionTree.
The associated implications of corresponding decision
gates are put in
this->backtrackImplicatedGateIDs_.
When we backtrack a single gate in decision tree, we
need to recover all
the associated implications starting from the startPoint
in
this->backtrackImplicatedGateIDs_.
                      +------+------+
 DecisionTreeNodes  : | G0=1 | G1=0 |...
                      +------+------+
      startPoint of G0=1 |        \ startPoint of G1=0
                         V         \
           .
                      +----+----++----+----+----+
 backtrackList:       |G4=0|G7=1||G6=1|G9=0|G8=0|...
                      +----+----++----+----+----+
                      \____ ____/\             /
                           V      \_____ _____/
associated implications of G0=1         V
                         associated implications of G1=0
```

Arguments	[in, out] `backwardImplicationLevel`: The backward implication level of current single pattern generation, will be updated in this function if any backtrack happened.
Output	A boolean indicating whether backtrack has failed, if failed the whole current target fault is determined as untestable in this atpg algorithm.

bool Atpg::continuationMeaningful(Gate *pLastDFrontier)

Synopsis	Used in single pattern generation to see if it is meaningful to continue.
Description	First call updateUnjustifiedLines(). If any gate in this->initialObjectives_ is modified, pop it from this->initialObjectives_.

	If the atpgVal_ of last D-Frontier has changed or all init objectives modified(Atpg::initialObjectives_.empty()), there is no need to continue doing the backtrace based on the current status.
Arguments	[in] `pLastDFrontier`: The last d-frontier in this->dFrontiers
Output	A boolean indicating if continuation in atpg is meaningful.

void Atpg::updateUnjustifiedGateIDs()

Synopsis	Update this->unjustifiedGateIDs_.
Description	Traverse all gates in this->unjustifiedGateIDs_, if any gate was put into unjustified list but was implied afterwards by other gates, remove those gates from this->unjustifiedGateIDs_. IF modifiedGate is modified, Delete it from this->unjustifiedGateIDs_ Else, Push modifiedGate into this->finalObjectives
Arguments	void
Output	void

void Atpg::updateDFrontiers()

Synopsis	Update this->dFrontiers_.
Description	Remove determined d-frontiers and add new propagated d-frontiers into this->dFrontiers_.
Arguments	void
Output	void

bool Atpg::checkIfFaultHasPropagatedToPO(bool &faultHasPropagatedToPO)

Synopsis	Check if fault has propagated to PO/PPO.
Description	If there is any D or B at PO/PPO, assign faultHasPropagatedToPO to true and return true. Otherwise assign false and return false.
Arguments	[in, out] faultHasPropagatedToPO: Will be assigned to true if the fault has propagated to PO/PPO.
Output	A boolean value same to faultHasPropagatedToPO

bool Atpg::checkForUnjustifiedBoundLines()

Synopsis	Check for any left unjustified bound lines.
Description	none
Arguments	void
Output	A boolean indicating if any unjustified bound lines are left in current single pattern generation.

void Atpg::findFinalObjective(BACKTRACE_STATUS &backtraceFlag,
const bool &faultCanPropToPO,
Gate *&pLastDFrontier)

Synopsis	Determination of final objectives.
Description	Choose a value and a line such that if the chosen value is assigned to the chosen line the initial objectives will be satisfied.
Arguments	[in, out] `backtraceFlag`: It indicates the backtrace atpgStatus. [in] `faultCanPropToPO`: It indicates whether the fault signal can propagate to PO/PPO or not.

	[in, out] `pLastDFrontier`: A pointer reference of pointer of the last d-frontier in single pattern generation.
Output	void

void Atpg::clearAllObjectives()

Synopsis	Clear and reinitialize all the objectives.
Description	none
Arguments	void
Output	void

void Atpg::assignAtpgValToFinalObjectiveGates()

Synopsis	Literal meaning of this function name.
Description	Decide the atpgVal_ of final objective gates by n0 and n1 calculated by previous multiple backtrace.
Arguments	void
Output	void

void Atpg::justifyFreeLines(const Fault &originalFault)

Synopsis	Justify free lines before terminating current single pattern generation.
Description	None
Arguments	[in] `originalFault`: The original target fault for single pattern generation.
Output	void

void Atpg::restoreFault(const Fault &originalFault)

Synopsis	Restore the faulty gate to the original position.
Description	This function is called because when the original target fault is injected at an gate input, it will then be modified to equivalent headline fault and set to the corresponding gate.atpgVal_. We need to the revert the previously mentioned operation for latter algorithm in atpg.
Arguments	[in] `originalFault`: The original target fault for single pattern generation.
Output	void

int Atpg::countEffectiveDFrontiers(Gate *pFaultyLineGate)

Synopsis	Update the this->dFrontiers to make sure the d-frontiers in it are all effective.
Description	By effective we mean if a d-frontier is able to propagate to PO/PPO.
Arguments	[in] `pFaultyLineGate`: The original target fault for single pattern generation.
Output	The updated this->dFrontier.size().

int Atpg::doUniquePathSensitization(Gate &gate)

Synopsis	Finds the last gate(pNextGate) in the uniquePath starting from pGate, return backwardImplicationLevel, which is the max of the pNextGate's input level. backwardImplicationLevel is -1 if no uniquePath.
Description	First check whether "gate" is the current target fault's gate. If not, we check the values of its fanin gates. If the value is unknown, set it to non-control value, push it into the this->backtrackImplicatedGateIDs, and call pushInputEvent function for this fanin gate. backwardImplicationLevel is set

	to the max of fanin level. If the value is control value, return UNIQUE_PATH_SENSITIZE_FAIL (-2). Now set the current gate to the input gate and enter the while loop. If the current gate is PO/PPO, or it has no dominators(excluded fanout free), leave the loop. Then set the next gate to the fanout gate(for fanout free gate) or its dominator. Now check whether the non-control value is not unknown and the gate is not unary. If false, do nothing, else we have two cases: fanout-free : Check the fanin gates of the next gate that is not the current gate. If its value is the control value, then return UNIQUE_PATH_SENSITIZE_FAIL. Otherwise if its value is unknown, set it to non-control value, push it into this->backtrackImplicatedGateIDs_. BackImpLevel is set to the maximum of fanin level. not fanout-free : Check the fanin gates of the next gate. If it is fault reachable, do nothing else check its value. If its value is the control value, return UNIQUE_PATH_SENSITIZE_FAIL. Otherwise if its value is unknown, set it to non-control value, push it into the backtrackList_, BackImpLevel is set to the maximum of fanin level. Finally, set the current gate to the next gate and start a new loop. Return BackImpLevel at last.
Arguments	[in] `gate` : The gate to do unique sensitization on.
Output	int(backwardImplicationLevel)

bool Atpg::xPathExists(Gate *pGate)

Synopsis	Determine if xpath exist for "gate".
Description	Used before generateSinglePatternOnTargetFault Return true if there is X-path. Otherwise return false.
Arguments	[in] `gate`: The gate to see if xpath exists.
Output	A boolean indicating if the x path exists.

bool Atpg::xPathTracing(Gate *pGate)

Synopsis	Determine if xpath exist for "gate".
Description	Recursive call the function itself with the fanout of original pGate until PO/PPO is reached and check if pGate has a X path.
Arguments	[in] `gate`: The gate to see if xpath exists.
Output	A boolean indicating if the x path exists.

int Atpg::setFaultyGate(Fault &fault)

Synopsis	Initial assignment of fault signal.
Description	There are two situations : 1. Fault is on the input line of pFaultyGate, and pFaultyLineGate is the fanin gate of pFaultyGate (1) Activate the fault, and set value of pFaultyLineGate according to fault

	type. (2) According to the type of pFaultyGate, set other fanin gate of pFaultyGate to NoneControl value of pFaultyGate, and set value of pFaultyGate. (3) Schedule all fanout gates of fanin gates of pFaultyGate, and schedule fanout gates of pFaultyGate. (4) Update backwardImplicationLevel to be max level of fanin gates of pFaultyGate. 2. Fault is on the ouput line of pFaultyGate, and pFaultyLineGate is pFaultyGate. (1) Activate the fault, and set value of pFaultyLineGate according to fault type. (2) Schedule fanout gates of pFaultyGate. (3) If pFaultyGate is a HEADLINE, all it's fanin gates are FREE_LINE, no need to set value. Else, set the value of it's fanin gates, and schedule all fanout gates of fanin gate of pFaultyGate. (4) Update backwardImplicationLevel to be max level of fanin gates of pFaultyGate.
Arguments	[in] `fault`: The fault for setting value to gate.
Output	The backwardImplicationLevel which indicates the backward imply level, return -1 when fault FAULT_UNTESTABLE

Fault Atpg::setFreeLineFaultyGate(Gate &gate)

Synopsis	Set equivalent fault according to the faulty gate.
Description	This function is called when gate is FREE_LINE. That means it has only one output gate. The returned fault must be on the output line of its gateID. In the while loop, sets unknown fanin gate of pCurrentGate to non-control value of pCurrentGate and sets the value of pCurrentGate. The loop breaks when pCurrentGate becomes a HEADLINE. When pCurrentGate is a HEADLINE, this function schedules all fanout gate of pCurrentGate, and decides the new fault type according to the value of pCurrentGate and returns the new fault.
Arguments	[in] `gate`: The faulty gate.
Output	The new head line fault that is equivalent to the original free line fault.

void Atpg::fanoutFreeBacktrace(Gate *pGate)

Synopsis	Backtrace in fanout free situation.
Description	none
Arguments	[in] `pGate`: The gate to start fanout free backtrace.
Output	void

Atpg::BACKTRACE_RESULT
Atpg::multipleBacktrace(BACKTRACE_STATUS atpgStatus, int &possibleFinalObjectiveID)

Synopsis	return NO_CONTRADICTORY or CONTRADICTORY after backtrace see paper P.4 P.5 and Fig.8 for detail information
Description	none

Arguments	[in] `atpgStatus`: it have 2 possibilities, atpgStatus == INITIAL means Multiple Backtrace from the set of initial objectives atpgStatus == FAN_OBJ_DETERMINE means Multiple Backtrace from the set of Fanout-Point Objectives [in, out] `possibleFinalObjectiveID`: Reference of possible fanout objective in single pattern generation.
Output	BACKTRACE_RESULT return CONTRADICTORY when we find a Fanout-Point Objective that is not reachable from the fault line and n0, n1 of it are both not zero; Otherwise, return NO_CONTRADICTORY n0 is the number of times objective 0 is required, n1 is the number of times objective 1 is required

Value Atpg::assignBacktraceValue(int &n0, int &n1, const Gate &gate)

Synopsis	Get n0, n1 and Value depending on Gate's controlling value.
Description	none
Arguments	[in, out] `n0`: n0 (int reference) to be set [in, out] `n1`: n1 (int reference) to be set [in] `gate`: gate to assign backtrace value to but the gate.atpgVal_ is assigned outside this function
Output	The decided value to assign to gate.

void Atpg::initializeForMultipleBacktrace()

Synopsis	Initialize all this->gateID_to_n0_, this->gateID_to_n1_.
Description	Copy the initial objectives into current o Traverse all gate in current objectives. If gate's atpgVal_ is L or B n0 = 1, n1 = 0 Else if gates's atpgVal_ is H pr D n1 = 0, n0 = 1 Else if X, Z, I Set line number depend on gate type
Arguments	void
Output	void

Gate *Atpg::findEasiestFaninGate(Gate *pGate, const Value &atpgValOfpGate)

Synopsis	Find the easiest fanin by gate::cc0_ or gate::cc1_.
Description	Utilize SCOAP heuristic if addSCOAP is called in setupCircuitParameter(). Otherwise cc0_ and cc1_ is 0. SCOAP heuristic is finished and can be found in atpg.cpp, but is not included in the algorithm because the result of SCOAP is even worse.
Arguments	[in] `pGate`: The gate to find easiest fanin gat. [in, out] `atpgValOfpGate`: The value to of the pGate.
Output	The easiest fanin gate to assign value.

Gate *Atpg::findClosestToPO(std::vector<int> &gateVec, int &index)

Synopsis	Find the gate which is the closest to output.

Description	none
Arguments	[in] `gateVec`: The gate vector to search. [in, out] `index`: The index of the gate closest to PO/PPO.
Output	The gate which is closest to PO/PPO.

Atpg::IMPLICATION_STATUS Atpg::evaluateAndSetGateAtpgVal(Gate *pGate)

Synopsis	The literal meaning of function name.

Description	If pGate is the faulty gate, return FaultEvaluation(pGate)
	else check the relationships between pGate's evaluated value and current value.
	If they are the same, set pGate to be modified,
	  return FORWARD,
	else if current value is unknown,
	  set it to the evaluated value and return FORWARD,
	else if the evaluated value is different from current value
	  return CONFLICT,
	else (only know current value)
	  return BackwardImplication(pGate).

Arguments	[in] `pGate`: The gate to do evaluation on.
Output	The implication status after this function call.

Atpg::IMPLICATION_STATUS Atpg::evaluateAndSetFaultyGateAtpgVal(Gate *pGate)

Synopsis	The literal meaning of function name.

Description	Check the relationships between pGate's current value and the evaluated value of pGate.
	If evaluated value is unknown
	  if pGate has current value,
	    if only one input has ONE unknown value
	      set the input to proper value and return BACKWARD
	    else
	      push pGate into unjustified_ list
	If they are the same
	  set pGate to be modified, return FORWARD
	If the evaluated value is different from current value
	  return CONFLICT

Arguments	[in] `pGate`: The gate to do evaluation on.
Output	The implication status after this function call.

void Atpg::staticTestCompressionByReverseFaultSimulation(PatternProcessor *pPatternProcessor, FaultPtrList &originalFaultList)

Synopsis	Perform reverse fault simulation to do static test compression.
Description	none
Arguments	[in, out] `pPatterProcessor`: The pattern processor contains the complete test pattern set before STC. It will then be reassigned to static compressed test pattern set. [in, out] `originalFaultList`: List of faults to be detected. Would be modified after this function call.
Output	void

inline void Atpg::writeAtpgValToPatternPI(Pattern &pattern)

Synopsis	Assign atpgVal_ of PI/PPI to PI/PPI in pattern.
Description	none
Arguments	[in, out] `pattern`: An empty pattern to be set.
Output	void

inline void Atpg::writeAtpgValToPatternPO(Pattern &pattern)

Synopsis	Assign atpgVal_ of PO/PPO to PO/PPO in pattern.
Description	none
Arguments	[in, out] `pattern`: An empty pattern to be set.
Output	void

B.3 Data structure

`CoreNS::Circuit`

This class stores information of the circuit, for example, gates in the circuit, number of gates, number of inputs, etc. As for the member functions, its main purpose is to parse the netlist and map it onto our data structure. You can build the circuit from netlist by calling `circuit::buildCircuit`.

bool Circuit::buildCircuit(Netlist *const pNetlist, const int &numFrame, const TIME_FRAME_CONNECT_TYPE &timeFrameConnectType)

Synopsis	Map the circuit to our data structure.
Description	We build the circuit with the input netlist. Also, determine the number of time frames and connection type for the circuit.
Arguments	[in] `pNetlist` : The netlist we build the circuit from. [in] `numFrame` : The number of time frames. [in] `timeFrameConnectType` : The connection type of time frames.
Output	bool : Indicate that we have constructed the circuit successfully.

Gate layout in Circuit:

```
frame 1                                  frame 2
|----- ------ ------ ----- ------|----- ------ ------
| PI1 | PPI1 | gate | PO1 | PPO1 | PI2 | PPI2 | gate | …
|----- ------ ------ ----- ------|----- ------ ------
```

Circuit gate information

`pCircuit_->gates_[j].goodSimLow_` and `pCircuit_->gates_[j].goodSimHigh_` store the value after Logic Simulation.
`pCircuit_->gates_[j].faultSimLow_` and `pCircuit_->gates_[j].faultSimHigh_` store the value after Fault Simulation.

The values are 2-bit encoded

```
cir_->gates_[j].goodSimLow_ = 00001110
cir_->gates_[j].goodSimHigh_= 00110001
```

It stands for eight pattern values, pattern 1 is X, pattern 2 is X, pattern 3 is 1, pattern 4 is 1, pattern 5 is 0, and so on.

`CoreNS::Pattern`

This class stores the information of a single pattern.

About the data type to store the pattern – `Value`

It's actually an unsigned 8-bit integer

```
const   Value   L       = 0;      Low
const   Value   H       = 1;      High
const   Value   X       = 2;      Unknown
const   Value   D       = 3;      D (good 1 / faulty 0)
const   Value   B       = 4;      D-bar (good 0 / faulty 1)
const   Value   Z       = 5;      High-impedance
const   Value   I       = 255;    Invalid
```

void Pattern::initForTransitionDelayFault(Circuit *pCircuit)

Synopsis	Basic setup initialization.
Description	Resize vectors PI2_ and PO2_ to numPI_ and numPO_ of the Circuit, and resize vector SI_ to 1.
Arguments	[in] `pCircuit` : The pointer to the target Circuit.
Output	void

`CoreNS::PatternProcessor`

This class stores a collection of patterns. It also has some information of the patterns like number of inputs and outputs, and the states that indicate whether the patterns are compressed.

The pattern processor supports two operations on the patterns:

(1) Static compression

(2) X-filling

void PatternProcessor::init(Circuit *pCircuit)

Synopsis	Basic setup initialization.
Description	1. Set numPI_, numPO_ and numPPI_ from the Circuit. 2. Set pPIorder_, pPPIorder_ and pPOorder_ = {0, 1,..., numPI_-1}. 3. Increase each element in pPOorder_ by (number of Gate in Circuit - numPO_ - numPPI_). 4. Increase each element in pPPIorder_ by numPPI_.
Arguments	[in] `pCircuit` : The pointer to the target Circuit.
Output	void

inline void PatternProcessor::StaticCompression()

Synopsis	Do static compression.
Description	Compare each pair of the patterns and check whether they are compatible. (i.e. can be merged without value assignment conflict) If so, merge these patterns bit by bit. The rule of compression: (X,L) -> L, (X,H) -> H
Arguments	void
Output	void

inline bool PatternProcessor::updateTable(std::vector<bool> mergeRecord, std::vector<bool> patternTable)

Synopsis	Function called in StaticCompression(). Try merging patterns according to the information given in the two input

	arguments.
Description	First store each pair of compatible patterns, and calculate their similarity. If no patterns can be merged, break and return false. Each time try merging the pairs with max similarity and update the mergeRecord and patternTable. Repeat the procedure until there are no remaining candidates.
Arguments	[in] `mergeRecord` : Stores whether each pattern can be merged. [in] `patternTable` : Stores whether each pair of patterns can be merged.
Output	[out] bool : true if patterns merged successfully, false if patterns can't be compressed correctly.

`CoreNS::PatternReader`

The class has two data members, `pCircuit_` and `pPatternProcessor_`, which are used to store the circuit and the patterns. The main purpose of the class is to parse the input patterns and save them into our data model.

The structure of the pattern file:

```
pattern_#    PI1 | PI2 | PPI1 | PPI2 | PO1 | PO2 | PPO
```

E.g.

```
_pattern_1  1101 |     |  010 |      |   1 |     | 101
```

void PatternReader::setPiOrder(const PatNames *const pPIs)

Synopsis	Map the PI order to the circuit order.
Description	First traverse all PIs to calculate the number of PIs, then set the order of PIs of the PatternProcessor according to the gate id of the circuit. The result will be stored in the vector pPIorder_ of the PatternProcessor.
Arguments	[in] `pPIs` : A pointer to the linked structure of primary inputs.
Output	void

void PatternReader::setPpiOrder(const PatNames *const pPPIs)

Synopsis	Map the PPI order to the circuit order.
Description	First traverse all PPIs to calculate the number of PPIs, then set the order of PPIs of the PatternProcessor according to the gate id of the circuit. The result will be stored in the vector pPPIorder_ of the PatternProcessor.
Arguments	[in] `pPPIs` : A pointer to the linked structure of pseudo primary inputs.
Output	void

void PatternReader::setPoOrder(const PatNames *const pPOs)

Synopsis	Map the PO order to the circuit order.
Description	First traverse all POs to calculate the number of POs, then set the order of POs of the PatternProcessor according to the gate id of the circuit. The result will be stored in the vector pPOorder_ of the PatternProcessor.
Arguments	[in] `pPOs` : A pointer to the linked structure of primary outputs.
Output	void

void PatternReader::setPatternType(const PatType &patternType)

Synopsis	Set the type of the Pattern Processor.
Description	Set type_ of the Pattern Processor according to the input patternType.

	If type is LAUNCH_SHIFT, set numSI_ to be 1.
Arguments	[in] `patternType` : Pattern Type to be set to.
Output	void

void PatternReader::setPatternNum(const int &patternNum)

Synopsis	Set the pattern vector according to the given size.
Description	Set each element of patternvector_ of the Pattern Processor to be default Pattern() with given input size (patternNum).
Arguments	[in] `patternNum` : Pattern number to be set.
Output	void

void PatternReader::addPattern(const char *const pPI1, const char *const pPI2,
const char *const pPPI, const char *const pSI,
const char *const pPO1, const char *const pPO2,
const char *const pPPO)

Synopsis	Read in a pattern and assign according values.
Description	For each input argument, assign according values to the corresponding vector of the Pattern Processor if it exists.
Arguments	[in] `pPI1` : The pointer to the first primary input pattern. [in] `pPI2` : The pointer to the second primary input pattern. [in] `pPPI` : The pointer to the pseudo primary input pattern. [in] `pSI` : The pointer to the shift in pattern. [in] `pPO1` : The pointer to the first primary output pattern. [in] `pPO2` : The pointer to the second primary output pattern. [in] `pPPO` : The pointer to the pseudo primary output pattern.
Output	void

void PatternReader::assignValue(std::vector<Value> &valueVector,
const char *const pattern, const int &size)

Synopsis	Set the pattern vector according to the given size.
Description	For each bit in the range of input size, assign value to the Value vector according to the input pattern content.
Arguments	[in, out] `valueVector` : The Value vector to be modified. [in] `pattern` : The pattern content to be assigned to. [in] `size` : The length of the input pattern.
Output	void

`CoreNS::PatternWriter`

The class has five member functions, four of which can dump the patterns into one distinguishing format, and one outputs the basic setup information.

(1) Lin Hsio-Ting's pattern format => no longer supported
(2) LaDS's own *.pat pattern format => support at most 2 time frames
(3) ASCII format
(4) STIL format

bool PatternWriter::writePattern(const char *const fname)

Synopsis	Write to LaDS's own *.pattern pattern format.
Description	Output the pattern to the given input file name with LaDS's own *.pattern

	pattern format. Support at most 2 time frames.
Arguments	[in] `fname` : The file name to be written to.
Output	bool : Output written successfully or not.

bool PatternWriter::writeLht(const char *const fname)

Synopsis	Write to Lin Hsio-Ting's pattern format.
Description	Output the pattern to the given input file name Ling Hsio-Ting's pattern format. Not supported now.
Arguments	[in] `fname` : The file name to be written to.
Output	bool : Output written successfully or not.

bool PatternWriter::writeAscii(const char *const fname)

Synopsis	Write to Mentor ASCII pattern format.
Description	Output the pattern to the given input file name with Mentor ASCII pattern format. Should be tested with Mentor Fastscan.
Arguments	[in] `fname` : The file name to be written to.
Output	bool : Output written successfully or not.

bool PatternWriter::writeSTIL(const char *const fname)

Synopsis	Write to STIL pattern format.
Description	Output the pattern to the given input file name with STIL pattern format. Should be tested with Tetramax.
Arguments	[in] `fname` : The file name to be written to.
Output	bool : Output written successfully or not.

bool PatternWriter::writeProcedure(const char *const fname)

Synopsis	Write the procedure setup information.
Description	Output the procedure setup to the given input file name, including time scale, strobe_window time, timeplate _default_WFT_ etc.
Arguments	[in] `fname` : The file name to be written to.
Output	bool : Output written successfully or not.

`CoreNS::Fault`

This class stores information of a single fault, including its type, state, and the gate which the fault is on.

Fault types

```
SA0    stuck-at zero
SA1    stuck-at one
STR    slow to rise
STF    slow to fall
BR     bridge
```

Fault states

```
UD     undetected
DT     detected
PT     possibly testable
AU     ATPG untestable
```

```
TI     tied to logic zero or one
RE     redundant
AB     aborted
```

`CoreNS::faultListExtract`
This class can store a list of faults that is extracted from the circuit, the function `FaultListExtract::extract()` is used to extract the faults.

void FaultListExtract::extractFaultFromCircuit(Circuit *pCircuit)

Synopsis	Extract faults from the circuit
Description	This function extracts uncollapsed faults, and extracts collapsed faults if needed. The method we use in fault collapsing is Simple Equivalent Fault Collapsing. In addition, we calculate the number of equivalent faults to recover the original uncollapsed fault coverage.
Arguments	[in] `pCircuit` : The circuit we want to extract faults from.
Output	void

`CoreNS::Gate`
The class stores all the information of a single gate. Also, the class provides functions that can determine the controlling and non-controlling value of the gate according to its gate type.

Gate types that are supported:

1) Input and output
   `PI, PO,PPI, PPO, PPI_IN, PPO_IN`

2) Logic gates
   ```
   INV, BUF,
   AND2, AND3, AND4, NAND2, NAND3, NAND4,
   OR2, OR3, OR4, NOR2, NOR3, NOR4,
   XOR2, XOR3, XNOR2, XNOR3
   ```

3) Others
   `MUX, TIE0, TIE1, TIEX, TIEZ`

inline Value Gate::isUnary() const

Synopsis	Check if the gate has only one fanin.
Description	If the gate has exactly one fanin, return H (Value). Otherwise, return L.
Arguments	void
Output	Value : Return H if the gate has exactly one fanin, return L otherwise.

inline Value Gate::isInverse() const

Synopsis	Check if the gate is an inverse gate.
Description	If the gate type is INV, NAND, NOR or XNOR, then it is an inverse gate. Otherwise, it is not.
Arguments	void
Output	Value : Return H if it is inverse gate, return L otherwise.

inline Value Gate::getInputNonCtrlValue() const

Synopsis	Get input non-control value of the gate.
Description	Determined by comparing the output of isInverse() and the output control

	value of the gate. If identical then the input non-control value is L. Otherwise it is H. If the gate type is INV, NOR or OR, input non-control value is L.
Arguments	void
Output	Value : Return the input non-control value of the gate.

inline Value Gate::getInputCtrlValue() const

Synopsis	Get input control value of the gate.
Description	Call getInputNonCtrlValue() to get input non-control value and return the inverse value.
Arguments	void
Output	Value : Return the input control value of the gate.

inline Value Gate::getOutputCtrlValue() const

Synopsis	Get output control value of the gate.
Description	If the gate type is OR or NAND, output control value is L. If the gate type is XOR or XNOR, output control value is X. Otherwise, output control value is H.
Arguments	void
Output	Value : Return the output control value of the gate.

`CoreNS::DecisionTree`

The class implements the decision tree that is used when we are doing backtracking in the FAN algorithm.

inline void DecisionTree::clear()

Synopsis	Clear tree_ of the DecisionTree.
Description	tree_ is a vector storing all the DecisionTreeNodes of the DecisionTree. Clear the vector tree_ using tree_.clear().
Arguments	void
Output	void

inline void DecisionTree::put(const int &gateId, const int &startPoint)

Synopsis	Add a new DecisionTreeNode to the tree_ of DecisionTree.
Description	Create a new DecisionTreeNode initiated with given arguments, and push the node to the back of the tree_ vector.
Arguments	[in] `gateId` : The gate Id to be assigned to the new DecisionTreeNode. [in] `startPoint` : The startPointInBacktrackImplicatedGateIDs_ to be assigned to the new DecisionTreeNode.
Output	Void

inline bool DecisionTree::get(int &gateId, int &startPoint)

Synopsis	Get the gateId and startPoint of the last DecisionTreeNode and check if it is marked.
Description	Find the last DecisionTreeNode of tree_ and: 1. Assign its gateId_ to the input argument gateId. 2. Assign its startPointInBacktrackImplicatedGateIDs_ to the input argument startPoint. 3. Return whether this node is marked (bool)
Arguments	[in, out] `gateId` : Will be assigned to the gate Id of the last DecisionTreeNode in tree_.

	[in, out] `startPoint` : Will be assigned to the class member startPointInBacktrackImplicatedGateIDs_of the last DecisionTreeNode in tree_.
Output	bool : Return whether the last DecisionTreeNode is marked.

inline bool DecisionTree::empty()

Synopsis	Check if the vector<DecisionTreeNode> tree_ is empty or not.
Description	Check if the vector tree_ is empty or not by using tree_.empty().
Output	bool : Return whether the tree_ of DecisionTree is empty.

inline bool DecisionTree::lastNodeMarked()

Synopsis	Check if the last element of tree_ is marked or not.
Description	First check if tree_ is empty. If yes, return false. Check if the last DecisionTreeNode in tree_ is marked or not. Used for backtracking in the algorithm.
Arguments	void
Output	bool : Return whether the last element of tree_ is marked.

`CoreNS::simulator`

The simulator is the class that controls and performs the flow of fault simulation.

inline void Simulator::setNumDetection(const int &numDetection)

Synopsis	Set number of detection (default = 1)
Description	Set numDetection_ (default = 1) for n-detect.
Arguments	[in] `numDetection` : The number of detection.
Output	void

inline void Simulator::goodSim()

Synopsis	Simulate the good value of every gate.
Description	Call the goodValueEvaluation function for each gate. Here we use goodSimLow_ and goodSimHigh_ instead of atpgVal_ in each gate.
Arguments	void
Output	void

inline void Simulator::goodSimCopyGoodToFault()

Synopsis	Simulate the good value of every gate and copy to fault value.
Description	Call the goodValueEvaluation function for each gate and copy the goodsim result to the faultsim variable.
Arguments	void
Output	void

inline void Simulator::goodValueEvaluation(const int &gateID)

Synopsis	Assign good value from fanin value to output of gate.
Description	Evaluate good output value (goodSimLow_ and goodSimHigh_) from the fanin values. We have the relationships : goodSimLow_ = 1, goodSimHigh_ = 0 => Real value = 0. goodSimLow_ = 0, goodSimHigh_ = 1 => Real value = 1. goodSimLow_ = 0, goodSimHigh_ = 1 => Real value = X.
Arguments	[in] `gateID` : The gate we want to evaluate.
Output	void

inline void Simulator::faultyValueEvaluation(const int &gateID)

Synopsis	Assign faulty value from fanin value to output of gate.
Description	Evaluate faulty output value (faultSimLow_ and faultSimHigh_) from the fanin values. We have the relationships : faultSimLow_ = 1, faultSimHigh_ = 0 => Real value = 0. faultSimLow_ = 0, faultSimHigh_ = 1 => Real value = 1. faultSimLow_ = 0, faultSimHigh_ = 1 => Real value = X. The calculation is similar to GoodValueEvaluation . The difference is that there are fault maskings at input and output of the gate.
Arguments	[in] `gateID` : The gate we want to evaluate.
Output	void

inline void Simulator::assignPatternToCircuitInputs(const Pattern &pattern)

Synopsis	Assign test pattern to circuit PI & PPI.
Description	Assign test pattern to circuit PI & PPI for further fault simulation.
Arguments	[in] `pattern` : The pattern we want to assign.
Output	void

void Simulator::eventFaultSim()

Synopsis	Do event-driven fault simulation.
Description	Call the faultyValueEvaluation function for gates in the event stacks and check if the faulty value is equal to the good value or not. If the values are the same, no more process is needed. If the values are not the same, keep processing.
Arguments	void
Output	void

void Simulator::parallelFaultFaultSimWithAllPattern(PatternProcessor *pPatternCollector, FaultListExtract *pFaultListExtract)

Synopsis	Perform parallel fault fault simulation on all patterns.
Description	First we extract undetected faults from the fault list. Then for each pattern, we assign the pattern and call the parallelFaultFaultSim function to do the fault simulation. May stop earlier if all faults are detected.
Arguments	[in] `pPatternCollector` : The patterns generated in ATPG. [in] `pFaultListExtract` : The whole fault list.
Output	void

void Simulator::parallelFaultFaultSimWithOnePattern(const Pattern &pattern, FaultPtrList &remainingFaults)

Synopsis	Perform parallel fault fault simulation on one pattern.
Description	Set the pattern and call the parallelFaultFaultSim function to do the fault simulation for this pattern.
Arguments	[in] `pattern` : The test pattern for the fault simulation. [in] `remainingFaults` : The list of undetected faults.
Output	void

void Simulator::parallelFaultFaultSim(FaultPtrList &remainingFaults)

Synopsis	Perform parallel fault fault simulation with assigned pattern.

Description	First we simulate good value. Then for the pattern, if the fault can be activated, inject the fault. When we inject enough faults or reach the end of the fault list, we run fault simulation for the injected faults and try to drop the detected faults. Here we can inject at most WORD_SIZE faults in one simulation.
Arguments	[in] `remainingFaults` : The list of undetected faults.
Output	void

void Simulator::parallelPatternGoodSimWithAllPattern(PatternProcessor *pPatternCollector)

Synopsis	Perform parallel pattern good simulation with all patterns.
Description	Set many patterns in parallel (at most WORD_SIZE) and run the good simulation.
Arguments	[in] `pPatternCollector` : The patterns generated in ATPG.
Output	void

void Simulator::parallelPatternFaultSimWithAllPattern(PatternProcessor *pPatternCollector, FaultListExtract *pFaultListExtract)

Synopsis	Perform parallel pattern fault simulation with all patterns on all faults.
Description	First we extract undetected faults from the fault list. Then we collect many patterns (at most WORD_SIZE) and call the parallelPatternFaultSim function to do the fault simulation on undetected faults for these patterns.
Arguments	[in] `pPatternCollector` : The patterns generated in ATPG. [in] `pFaultListExtract` : The whole fault list.
Output	void

void Simulator::parallelPatternFaultSim(FaultPtrList &remainingFaults)

Synopsis	Perform parallel pattern fault simulation on all faults after assigning patterns.
Description	First we simulate the good value for the assigned pattern. Then for all undetected faults, if the fault can be activated, inject the fault. If the fault can be detected, we can drop this fault.
Arguments	[in] `remainingFaults` : The list of undetected faults.
Output	void

void Simulator::parallelFaultReset()

Synopsis	Reset simulation after doing parallel fault fault simulation.
Description	Reset faulty value of the fault gate to good value. Also, reset processed flags and fault masks to 0.
Arguments	void
Output	void

bool Simulator::parallelFaultCheckActivation(const Fault *const pfault)

Synopsis	Check whether the fault can be activated to the fanout of the gate.
Description	Compare the goodSimLow_ & goodSimHigh_ of the faulty gate with the fault type to check whether the fault can be activated.
Arguments	[in] `pfault` : The fault we want to check.
Output	Bool : Indicate whether the fault can be activated or not. If activated then we can inject this fault.

void Simulator::parallelFaultFaultInjection(const Fault *const pfault, const size_t &injectFaultIndex)

Synopsis	Inject fault and push faulty gate into event list.
Description	In parallel fault fault simulation, we add fault on "one" bit in ParallelValue. Then we can have at most WORD_SIZE faults in one fault simulation.
Arguments	[in] `pfault` : The fault we want to inject. [in] `faultInjectIndex` : The index we want to inject to.
Output	void

void Simulator::parallelFaultCheckDetectionDropFaults(FaultPtrList &remainingFaults)

Synopsis	Check whether the injected fault can be detected by the pattern.
Description	Compare the result of the good simulator and fault simulator and check whether the injected fault can be detected by the pattern. Finally, drop the detected faults.
Arguments	[in] `remainingFaults` : The list of undetected faults.
Output	void

void Simulator::parallelPatternReset()

Synopsis	Reset simulation after doing parallel pattern fault simulation.
Description	Reset faulty value of the fault gate to good value. Also, reset processed flags, activated flags, and fault masks to 0.
Arguments	void
Output	void

bool Simulator::parallelPatternCheckActivation(const Fault *const pfault)

Synopsis	Check whether the fault can be activated to the fanout of the gate.
Description	Compare the goodSimLow_ & goodSimHigh_ of the faulty gate with the fault type to check whether the fault can be activated. We then save the activated flags for patterns in the activated_ variable.
Arguments	[in] `pfault` : The fault we want to check.
Output	Bool : Indicate whether the fault can be activated or not. If activated then we can inject this fault.

void Simulator::parallelPatternFaultInjection(const Fault *const pfault)

Synopsis	Inject fault and push faulty gate into event list.
Description	In parallel pattern fault simulation, we add fault on "all" bits in ParallelValue since we simulate the fault for all patterns.
Arguments	[in] `pfault` : The fault we want to inject.
Output	void

void Simulator::parallelPatternCheckDetection(Fault *const pfault)

Synopsis	Check whether the injected fault can be detected by the pattern.
Description	Compare the result of the good simulator and fault simulator and check whether the injected fault can be detected by the patterns. If yes, then set its fault state to detected(DT) for fault drop.
Arguments	[in] `pfault` : The fault we want to check.
Output	void

void Simulator::parallelPatternSetPattern(PatternProcessor *pPatternProcessor, const int &patternStartIndex)

Synopsis	Apply patterns to PIs and PPIs.
Description	Starting from PatternStartIndex, we apply patterns up to WORD_SIZE to PIs and PPIs for further fault simulation.
Arguments	[in] `pPatternProcessor` : The patterns generated in ATPG. [in] `PatternStartIndex` : Indicate where we start applying patterns in the pattern vector.
Output	void