SWiTEST: Program Organization and Manual

Kuen-Jong Lee and Melvin A. Breuer

CENG Technical Report 92-04

Department of Electrical Engineering – Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-4476

SWiTEST: Program Organization and Manual*

Kuen-Jong Lee and Melvin A. Breuer

May 5, 1992

*This work was supported by the Defense Advanced Research Projects Agency and monitored by the Federal Bureau of Investigation under Contract No. JFBI90092. The views and conclusions considered in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U. S. Government.

Contents

About this document

This document describes the program organization of SWiTEST, a switch level test generation system for CMOS circuits. The purpose of this document is to help the user understand the system architecture. Two companion documents, [1] and [2], should be read together with this document. Section 1 of this document gives a brief introduction to SWiTEST, Section 2 provides the block diagrams of the system, and Section 3 describes the function of each routine.

1 Introduction

SWiTEST is designed for generating tests for CMOS combinational circuits. The circuits that can be analysed by the system are described in [1] and [2]. These circuits can be roughly classified as fully complementary MOS combinational circuits. The faults that can be tested include transistor stuck-open and stuck-on faults, node stuck-at faults, and bridging faults between any two circuit nodes, where a node can be any I/O or internal node of a gate or transistor group.

SWiTEST consists of two major programs: `prep` and `switest`. `prep` is used for preprocessing a circuit so that useful information for test generation can be obtained. This information includes circuit partitioning, primitive and complex gate identification, signal flow direction of transistors, and the relative position of circuit partitions. Signal flow direction assignment forms a major part of this program. `switest` executes the actual test generation procedures. It employs a PODEM-based algorithm with a test generation framework containing such routines as objective selection, backtracing, logic implication, backtracking and fault propagation.

2 Block diagrams

The block diagrams of SWiTEST are shown in Figures 1–4. In Figures 1 and 3 each block contains the name of a procedure and the name of the file (in parentheses) that contains this

procedure. In Figure 2 only names of procedures are given since all procedures are in the same file. In Figure 4, in addition to the procedure and file names, the test generators that employ the procedures are also given.

3 Function of each routine

Preprocessor — `prep`

- `main`: main program of preprocessor.
- `cirin`: circuit input routine.
- `partition_and_direction`: partitions the circuit into transistor groups, identifies primitive and complex gates, and assigns signal flow direction to transistors.
- `level_and_order`: assigns input and output levels and topological sorting orders to transistor groups.
- `write_to_files`: generates two output files, one `*.tg` and another `*.lst`.
- `form_a_tg`: creates a transistor group.
- `identify_a_gate`: identifies gates from a transistor group.
- `direction`: assigns signal flow direction to transistors in a transistor group.
- `st_graph_gen`: generates an ST-graph for a given transistor group.
- `ps_reduction`: performs ps_reduction on an ST-graph. No direction assignment is done in this procedure.
- `assign_one`: If only one edge exists after PS-reduction, then this routine is called to assign direction to the edge.
- `assign_direction`: If more than one edge exist after PS-reduction, then this routine is called for assigning directions.

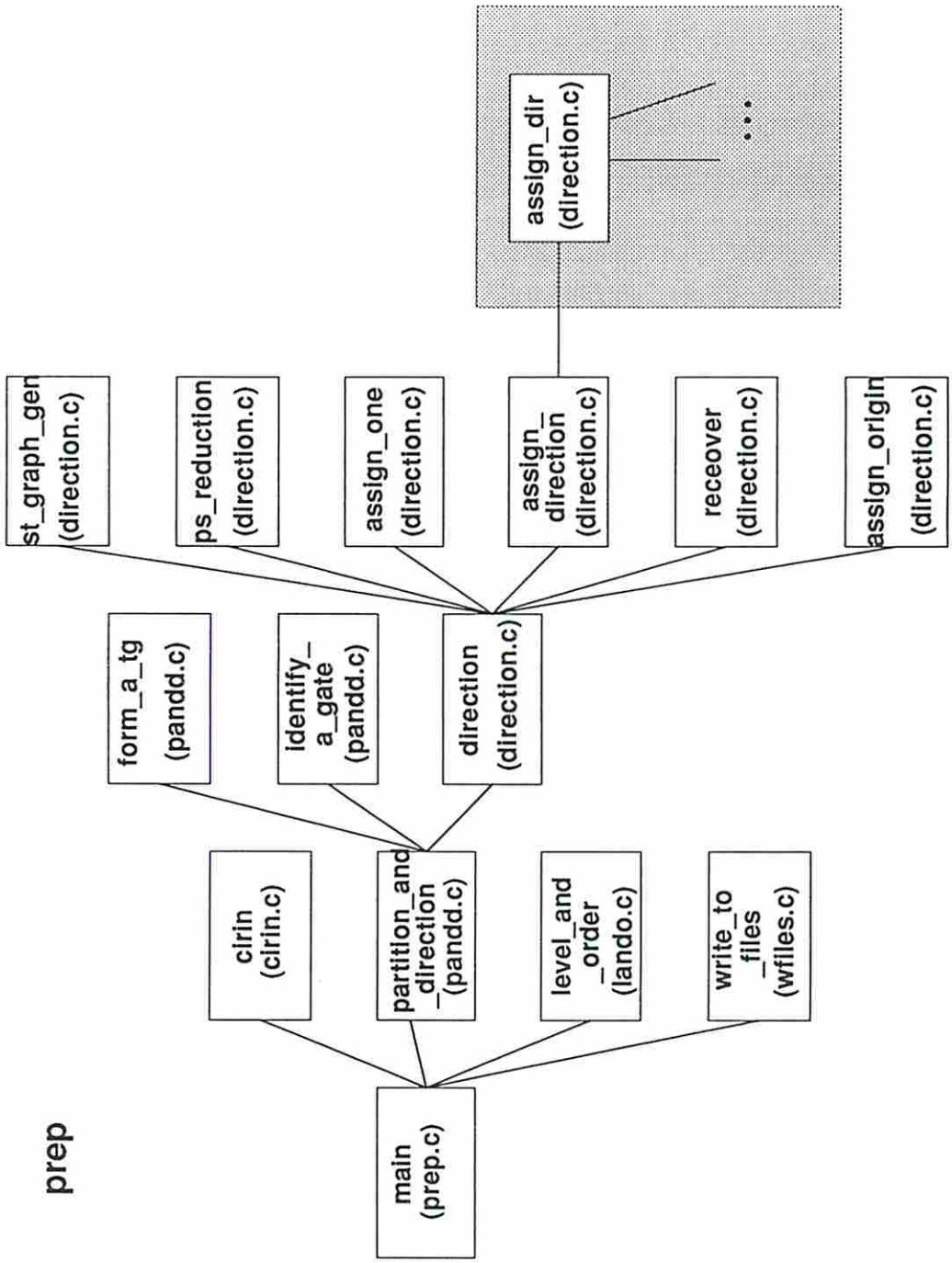


Figure 1: Preprocessor of SWiTEST

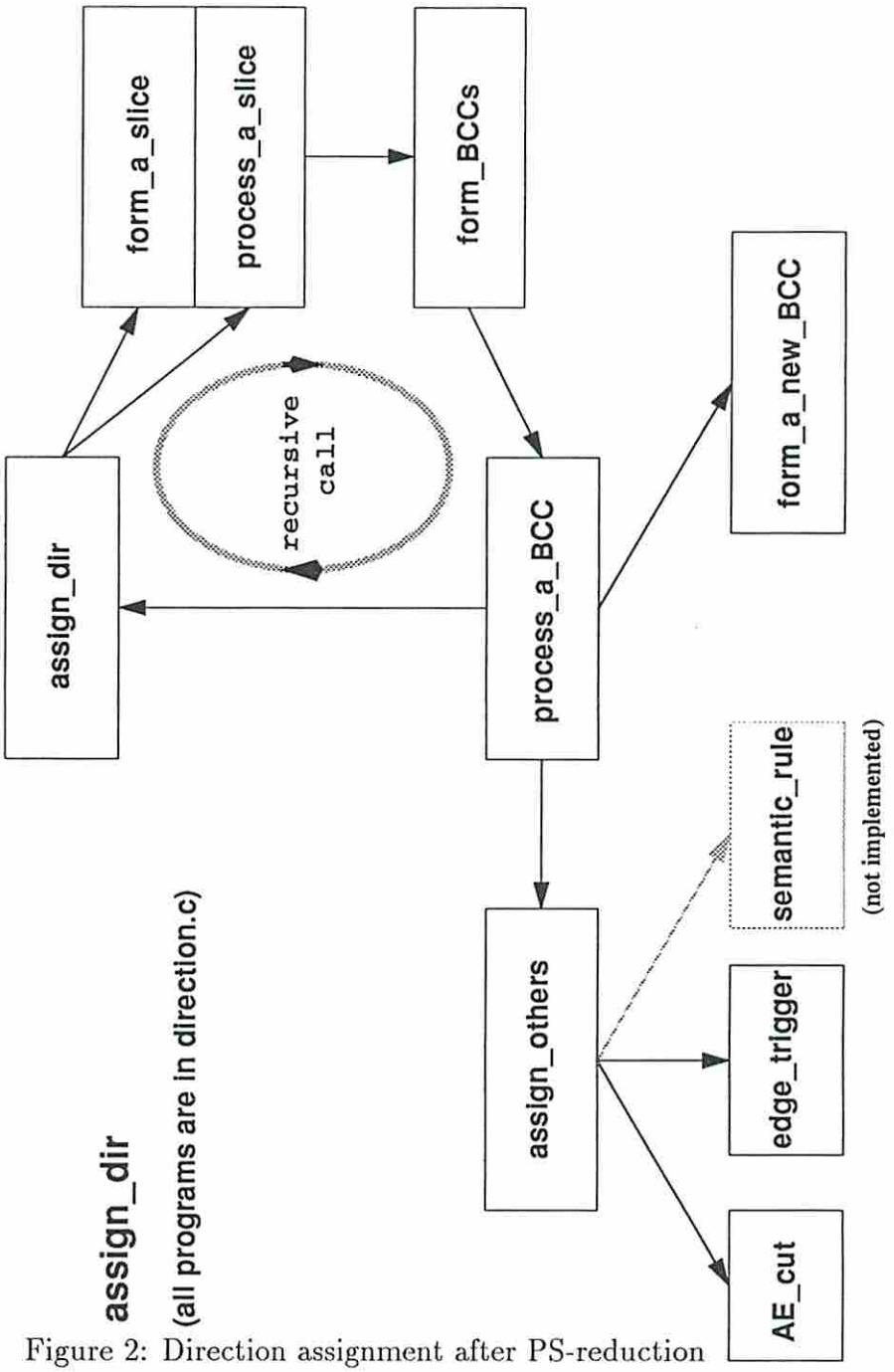


Figure 2: Direction assignment after PS-reduction

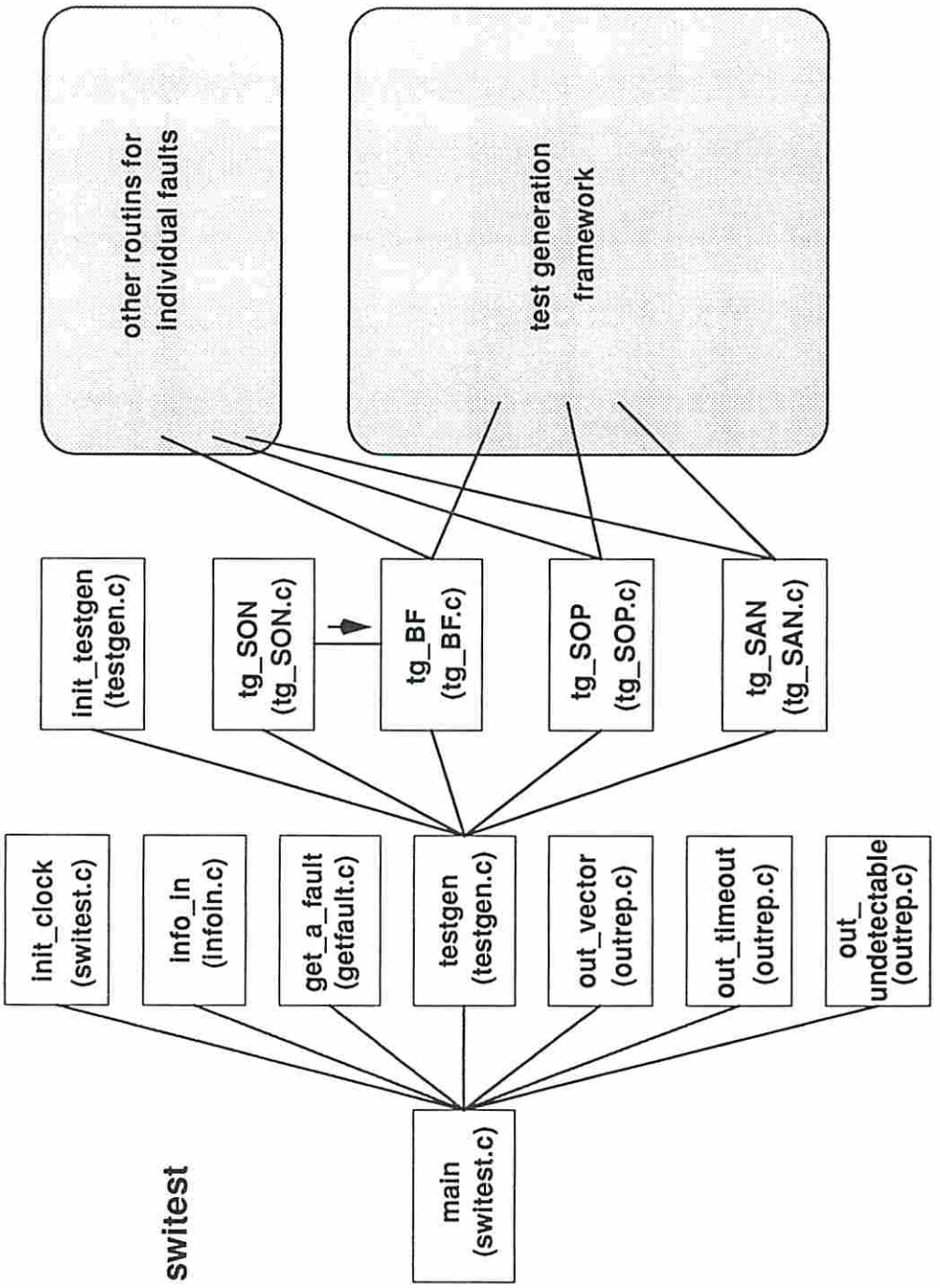


Figure 3: Test generation part of SWiTEST

Figure 4: Test generation framework

- `recover`: reverse procedure of PS-reduction. When an edge is recovered, its direction is also assigned.
- `assign_origin`: assigning direction to transistors in original circuits based on the direction information in the corresponding ST graph.

Direction assignment — `assign_dir`

- `assign_dir`: assigns directions to transistors when PS-reduction cannot reduce an ST-graph to one single edge. It is called by `assign_direction` or by `process_a_BCC` to initialize the process of a new slice.
- `form_a_slice`: obtains a slice from a reduced ST-graph.
- `process_a_slice`: processes a slice. This routine and `form_a_slice` are repeatedly called by `assign_dir` until all slices are processed.
- `form_BCCs`: identifies all BCCs in one slice.
- `process_a_BCC`: processes one BCC. If the BCC is indivisible, calls `assign_others` to assign directions in the BCC.
- `form_a_new_BCC`: creates new data structure for the BCC. This is necessary because of the recursive calls.
- `assign_others`: assigns directions to edges in an indivisible BCC.
- `AE_cut`: AE_cut routine.
- `edge_trigger`: performs two edge_trigger processes.
- `semantic_rule`: for expansion use. Not implemented yet.

Main program for test generation — `switest`

- `main`: main program for `switest`.

- `init_clock`: initializes the system clock. This is used to determine system performance.
- `info_in`: the input routine which reads in the required circuit information created by `prep`.
- `get_a_fault`: gets one new fault from the provided fault file.
- `testgen`: test generation for the given fault.
- `out_vector`: reports the test vector(s) when the fault is detected.
- `out_timeout`: reports that the backtracking limit is excessed.
- `out_undetectable`: reports that the fault is identified as undetectable.
- `init_testgen`: initializes the test generation process.
- `tg_SON`: test generation for a stuck-on fault. This procedure converts a stuck-on fault to a bridging fault and calls `tg_BF` for actual test generation.
- `tg_BF`: test generation for a bridging fault.
- `tg_SOP`: test generation for a stuck-open fault.
- `tg_SAN`: test generation for a node stuck-at fault.
- Test generation framework: the PODEM based test generation framework.
- Other routines for individual faults: routines which cannot be classified as those in the test generation framework.

Test generation framework

- `set_obj`: sets up an objective.
- `assign_obj`: determines which set of initial objectives to be used when dealing a bridging fault.

- `exchange`: exchanges the logic values to be assigned to the two shorted node when dealing with a bridging fault.
- `get_obj`: gets an objective from the cutset information when dealing with a stuck-open fault.
- `backtrace`: backtracing routine. Used when generating a test for a bridging fault or the first vector for stuck-open fault.
- `backtrace2`: backtracing routine. Used when generating the second vector for a stuck-open fault. Also used when generating stuck-at fault. The difference between `backtrace` and `backtrace2` is that `backtrace` is used only for the fault-free circuit, while `backtrace2` must deal with both the faulty and fault-free circuits.
- `backtrack`: backtracking routine. Used when dealing with a bridging fault.
- `backtrack0`: backtracking routine. Used when generating the transient vector to satisfy the cutset condition.
- `backtrack1`: backtracking routine. Used when generating the first test vector for a stuck-open fault.
- `backtrack2`: backtracking routine. Used when generating the second test vector for a stuck-open fault. Also used for stuck-at fault.
- `implication`: forward implication. Used for bridging fault.
- `implication0`: forward implication. Used when generating the transient vector for a stuck-open fault.
- `implication1`: forward implication. Used when generating the first test vector for a stuck-open fault.
- `implication2`: forward implication. Used for a fault-free circuit when generating the second vector for a stuck-open fault. Also used for a stuck-at fault.
- `implication2_bad`: forward implication. Used for a faulty circuit when generating the second vector for a stuck-open fault. Also used for a stuck-at fault.
- `find_obj_from_Dfront`: the main routine for fault propagation. Used for stuck-open and stuck-at faults.

References

[1] K.J. Lee. *Switch level test generation for CMOS circuits.* PhD thesis, Univ. of Southern California, Los Angeles, August 1991.

[2] K.J. Lee. SWiTEST User's Manual — Version 1.0. Technical Report CENG-91-??, Univ. of Southern California, Los Angeles, California, Aug. 1991.

SWiTEST User's Manual — Version 1.0 *

Kuen-Jong Lee

Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, CA 90089-2560

August 30, 1991

*This work was supported by the Defense Advanced Research Projects Agency and monitored by the Office of Naval Research under Contract No. N00014-87-K-0861, and by the Federal Bureau of Investigation under Contract No. JFBI90092. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U. S. Government.

Contents

About this manual

This manual describes the use of SWiTEST—a switch level test generation system for CMOS circuits. The organization of this manual is as follows. Section 1 contains general information about the system, which includes the scope and organization of the system and a brief description of each function provided. Section 2 summarizes the system environment and the input/output specifications. Default values of several system parameters such as the maximum number of transistors and the maximum number of transistors connected to a node are given. The exact format of each file used in the system is then described, followed by the input/output requirements and the user interface for each function. Section 3 illustrates the various aspects of the system through an example circuit. The main objective of this manual is to illustrate how to use SWiTEST. For detailed description about the theoretical background behind SWiTEST, the reader should refer to [Lee91].

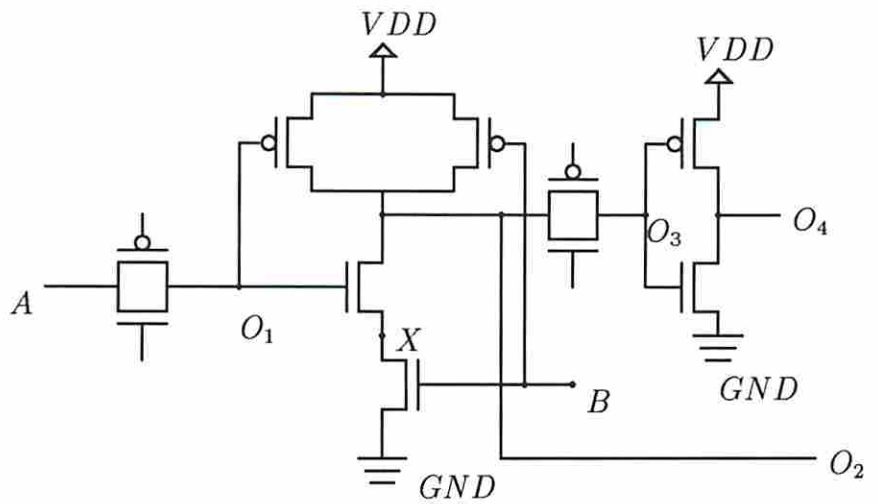
1 General description

1.1 Scope of system

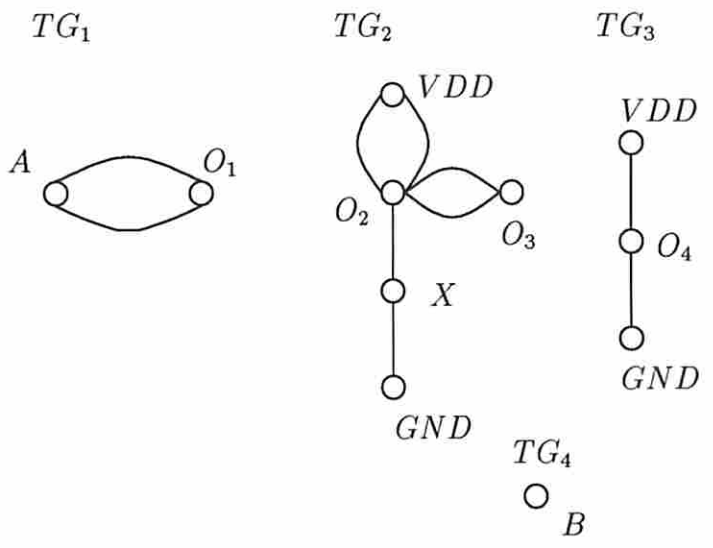
SWiTEST is designed for generating test vectors for CMOS circuits described at the switch level. Currently these circuits can be roughly classified as fully complementary MOS (FC-MOS) combinational circuits. The following description gives a precise definition of these circuits.

A switch level CMOS circuit consists of N-type/P-type transistors and nodes connecting these transistors. We may partition a CMOS circuit into a number of "channel-connected components" called *transistor groups(TGs)* such that any two transistors whose channels are connected through their drain or source terminals are in the same TG, except for the case where the connection between their channels is either *VDD* or *GND*. A primary input that is not connected to the drain or source of any transistor forms a *trivial* TG. Figure 1 shows the partition of a circuit and some of its *TGs*, where node B is a trivial TG. Each interconnection between *TGs* is unidirectional and can be considered as an *input* or *output* of the corresponding *TGs*. A transistor group *TA* can *control* another transistor group *TB* if one of the outputs of *TA* affects one of the inputs of *TB*. For example, in Figure 1 TG_1 can control TG_2 and TG_3, and TG_2 can control TG_3. If two TGs can control each other, then a *control loop* exists.

Depending on the faults to be detected, SWiTEST may employ logic monitoring or current sensing (IDDQ testing) or both as the fault effect observation method. The following constraints on a fault-free circuit and its test environment must be satisfied when both logic monitoring and IDDQ testing are used. If only logic monitoring is used, then A1, A5 and A6 can be released.



(a)



(b)

Figure 1: Partitioning of a CMOS circuit

A1. The gate and drain (or source) node of a transistor cannot be in the same *TG*.

A2. During steady state operation, there must be no conducting path from *VDD* to *GND*.

A3. During steady state operation, each output of a transistor group must be connected to *VDD* or *GND* through a path of conducting transistors.

A4. There are no control loops among *TGs*.

A5. The substrate (or well) of an N-type(P-type) transistor is connected to *GND(VDD)*.

A6. During testing, each primary input is controlled by a strong power source whose current flow is also monitored.

SWiTEST can generate tests for the following faults: all transistor stuck-open faults, all transistor stuck-on faults, any bridging fault between any two circuit nodes, and all node stuck-at faults, where a node can be either an input/output node or an internal node of a transistor group. SWiTEST uses a PODEM-like algorithm. It either identifies a fault as undetectable, finds a test for the fault, or aborts a fault after a given number of backtrackings is excessed.

SWiTEST may generate single- or double-vector tests. When dealing with stuck-at faults, it employs logic monitoring only, i.e., a fault is detected only when there exists a primary output whose logic values differ in the fault-free and faulty circuits. Also it only generates single vector tests. For a stuck-open fault SWiTEST assumes that a two-vector test is required. The first vector sets up an initial condition and the second vector propagates the fault effect to some primary output. Only tests that cannot be invalidated under any circuit delay are generated. It is assumed that charge sharing cannot invalidate a test when IDDQ testing is used. For bridging and stuck-on faults only IDDQ testing is employed. Since it is assumed that A1-A6 are satisfied, single vector tests are sufficient to detect any irredundant stuck-on or bridging fault.

Currently no fault collapsing or fault simulation is implemented.

1.2 System organization

The organization of SWiTEST is shown in Figure 2. The system contains three parts: a preprocessor, a test generation framework and a number of individual test generators for various fault models.

The preprocessor is used to extract useful circuit information for test generation. This information includes the partitioning of a circuit, identification of complex and primary gates, input/output levels of circuit partitions, and signal flow directions of transistors. The information obtained is stored in files so that they can be repeatedly used. The box containing

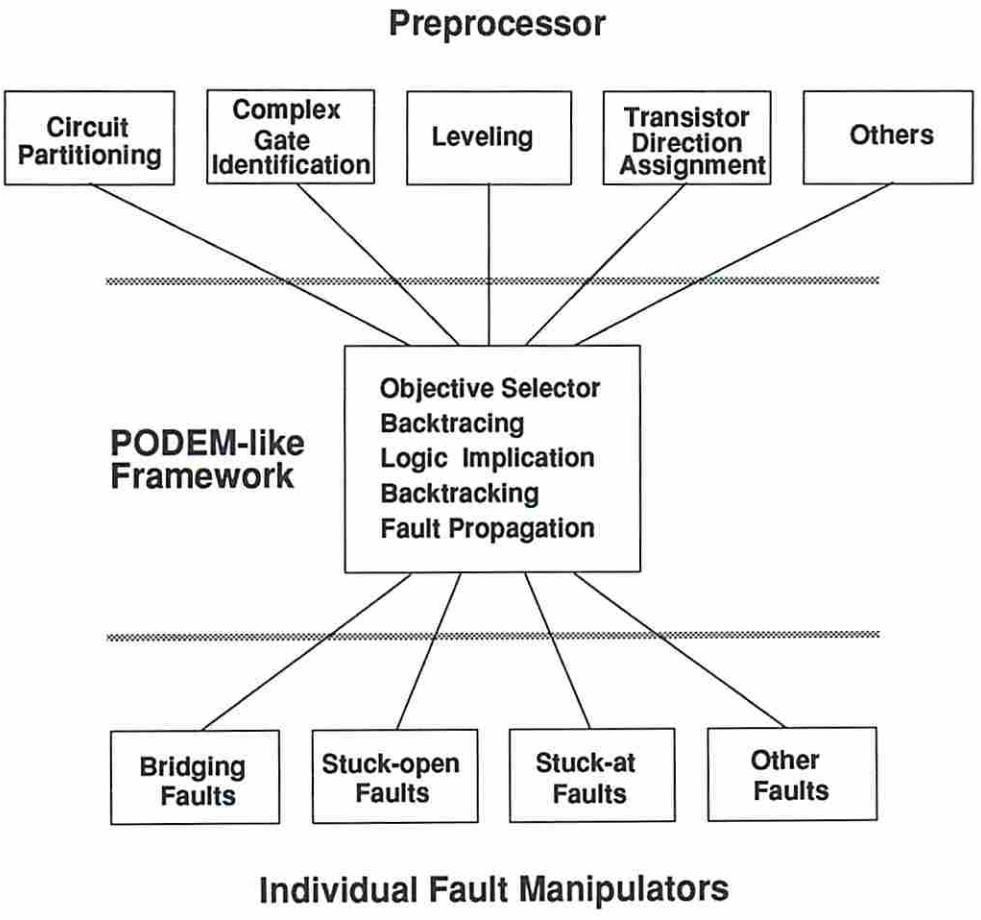


Figure 2: Organization of SWiTEST

"others" is for future extensions, e.g., to implement the concepts used in FAN or Socrates. The test generation framework is PODEM-based, which includes 5 major components: objective selection, backtracing, logic implication, backtracking and fault propagation. These components are implemented as modular routines. Each of these routines can be accessed by each individual fault manipulator. Three individual test generators, for bridging, stuck-open and stuck-at faults, are currently available. The system can also deal with any stuck-on fault by converting it to a bridging fault between the drain and source of the faulty transistor. It is also possible to deal with a line breaking fault by converting it to a stuck-at or stuck open fault though currently this is not implemented. In the future the system can be extended to include other faults such as delay faults and can be tied to a fault simulator so that a complete *automatic test pattern generation* system (ATPG) can be developed.

1.3 Functions

The central goal of SWiTEST is to generate tests for CMOS circuits at the switch level. To achieve this goal two major programs are implemented: `prep` and `switest`.

- `prep`: This program is used to preprocess a circuit and create the information needed for test generation. The input to `prep` is a circuit file with a spice-like format. In addition to the flat SPICE description of a circuit, two additional command lines `.INPUTS` and `.OUTPUTS` must be added to the input file to provide information on primary inputs and outputs.

  `prep` partitions the input circuit into transistor groups, identifies complex and primitive gates, assigns levels to each transistor groups and assigns signal flow directions to transistors. The information is written into two files: a file named `*.lst` which contains information about circuit nodes and transistors, and a file named `*.tg` which contains information about transistor groups and gates, where "*" is the name of the input circuit file.

- `switest`: After generating `*.lst` and `*.tg` files, `switest` can be invoked to generate test vectors for faults. The faults to be detected are listed in a "fault file" named "`*.flt`," where "*" is again the name of the input circuit file. Each fault in a fault file can be a stuck-open, bridging, stuck-on or stuck-at fault. `switest` processes the faults in the fault file one by one. A time-out mechanism based on the number of backtrackings is implemented. For each fault, `switest` either reports the test that detects the fault, declares the fault as undetectable, or aborts the test generation process after the number of allowed backtrackings is excessed.

To assist the generation of switch level circuits and fault lists, several programs are implemented. Also since the signal flow direction assignment work may be used in fields other than testing, a stand alone program called `dire` is implemented. These programs are listed below.

- `g2s`: A translator to translate a circuit with a "tgs" format to one with a spice-like format. A tgs format is a format for a gate level circuit, which is used in the USC Test Generation System, Version 1.1 [LB88].

- `bfgen`: An interactive program to help the user generate a file containing random bridging faults in a circuit.

- `asangen`: A program to generate all node stuck-at faults in a circuit. These include all stuck-at faults on both I/O and internal nodes of a transistor group.

- `sangen`: A program to generate all stuck-at faults at the outputs of transistor groups in a circuit.

- `songen`: A program to generate all transistor stuck-on faults in a circuit.

- `sopgen`: A program to generate all transistor stuck-open faults in a circuit.

- `es2s`: A special translator to translate a circuit with an "esim" format to one with a spice-like format. Currently this program is specially designed for the circuit "ald-chip.esim" which was provided by Rajiv Gupta at GE and contains more than 34,000 transistors. It takes about 50 minutes to translate the circuit format because of the "linear search" mechanism used to determine whether a node name has occurred before. This inefficiency can be improved by using a binary search tree mechanism.

  The program can be modified for different esim files. Such modification is necessary because each circuit has different names for its primary inputs and outputs.

- `dire`: A stand alone program for assigning signal flow directions to transistors.

2 I/O specifications

2.1 System environment

All programs are written in C and run on a SUN Workstation (SUN 4) under SunOS Release 4.1.1.

The data structures used in `prep` and `switest` are different. Most data structures used in `prep` are arrays with fixed sizes. The following are the current array sizes.

- maximum number of transistors: 20,000
- maximum number of nodes: 10,000
- maximum number of primary inputs: 1,000

- maximum number of primary outputs: 1,000
- maximum number of transistors per TG: 300
- maximum number of nodes per TG: 150
- maximum number of transistor groups: 7,000
- maximum number of complex gates: 7,000
- maximum number of input nodes to a TG: 100
- maximum number of output nodes of a TG: 100
- maximum number of transistors connected to a non-*VDD* or *GND* node: 150

If it is desired to run `prep` with different parameters, the user must modify these constants and re-compile the program. The user should make sure the above parameters are large enough for the circuit under test. However the program efficiency depends on the size of these parameters. Thus using large values of these parameters for a small circuit tends to reduce program performance.

In `switest` all the above data structures are automatically calculated and allocated in the initialization procedures. This is done by using the dynamic allocation facility of the C language.

2.2 File formats

Four types of files are used in SWiTEST. They respectively contain 1) the switch level description of the circuit to be tested, 2) information about nodes and transistors of the circuit, 3) information about transistor groups and gates, and 4) a list of faults to be tested. Next the formats of these files are described. The description for each type of file consists of three parts separated by a dashed line. The first part is the actual file format, the second part explains the file format and the third part gives an example.

In the file format, a line with arguments separated by "`|`" denotes that any of these arguments can be used in that line. An argument enclosed by a pair of "`[`" and "`]`" is an optional item. The characters between two double-quotes (such as `abc` in `"abc"`) are actual characters appearing in the file.

Input switch level circuit file

This file contains the switch level description of the circuit. It usually has a file name "`*.s`," where "`.s`" denotes that the file has a spice-like format.

```
------------------------------------------------------------
File format:

<Comment>
<comment> | <Restricted SPICE description>
<comment> | <Restricted SPICE description>
 ...
<comment> | <Restricted SPICE description>
".INPUTS" #PIs PI1 PI2 ...
".OUTPUTS" #POs PO1 PO2 ...
".END"
------------------------------------------------------------
File description:

Each comment line must start with a "*".
The first line is a comment line.

The third and second to last lines must start with .INPUTS and .OUTPUTS,
     respectively, to represent the primary inputs and outputs.  #PIS
     and #POS are the numbers of primary inputs and outputs, respectively.
The last line must be ".END".

Any other line is either a comment or a restricted SPICE format
     which must start with a character "M" or "C" to represent
     a MOS transistor or a capacitor.

The program will ignore a comment line or a line starting with C.

A line starting with an "M" has the following format:

    name drain gate source bulk model [length] [width] [other parameters]

  where name = name of the transistor (maximum 10 characters)
        drain, gate, source, bulk = the index (number) of the drain,
            gate, source, bulk terminals of the transistor, respectively;
        model = the model of the transistor. The first character of
            model determines the type of the transistor (P or N type);
        length, width = optional length and width of the channel;
        other parameters = optional circuit parameters.

The length, width and other parameters are ignored by all the currently
     available programs.
------------------------------------------------------------
```

```
Example: c60.s

*This file is generated by g2s. The input file is c60.
* Primary input one[1], node number = 2
* Primary input two[2], node number = 3
* Primary input three[3], node number = 4
* Primary input four[4], node number = 5
...
* Primary input b[14], node number = 15
* Primary input c[15], node number = 16
* Gate d[16]: 2-input nand, node number=18, inputs=[one,two]
MP1 18 2 1 1 PSS L=10U W=4U
MN1 17 2 0 0 NSS L=10U W=2U
MP2 18 3 1 1 PSS L=10U W=4U
MN2 18 3 17 0 NSS L=10U W=2U
* Gate e[17]: 2-input nand, node number=20, inputs=[two,three]
MP3 20 3 1 1 PSS L=10U W=4U
MN3 19 3 0 0 NSS L=10U W=2U
MP4 20 4 1 1 PSS L=10U W=4U
MN4 20 4 19 0 NSS L=10U W=2U
...
* Gate v[38]: 2-input nand, node number=71, inputs=[t,u]
MP54 71 67 1 1 PSS L=10U W=4U
MN54 70 67 0 0 NSS L=10U W=2U
MP55 71 69 1 1 PSS L=10U W=4U
MN55 71 69 70 0 NSS L=10U W=2U
*
.INPUTS 15 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
.OUTPUTS 4 56 58 65 71
.END
------------------------------------------------------------
```

Node and transistor list file

The node and transistor file, also called a "list file," usually has the file name "`*.lst`", where "`*`" is the circuit file name. For example if the circuit file name is "`c60.s`" then the list file is usually "`c60.s.lst`".

```
------------------------------------------------------------
File format:

<Comment>
```

```
#PI "PIS:" PI1 PI2 ...
#PO "POS:" PO1 PO2 ...
"The maximum node number is" #nodes
"0 GND" #pn_gnd #nn_gnd #gn_gnd
"1 VDD" #pn_vdd #nn_vdd #gn_vdd
node_num  node_name tg gate #pn #nn #gn is_pi is_po
 <transistor list>
node_num  node_name tg gate #pn #nn #gn is_pi is_po
 <transistor list>
  ...
node_num  node_name tg gate #pn #nn #gn is_pi is_po
 <transistor list>
#trans "transistors"
tran_num tran_name mos tg gate snode dnode gnode dir
tran_num tran_name mos tg gate snode dnode gnode dir
  ...
-------------------------------------------------------------
File description:

The first line is a comment line.
The second and third lines gives the primary input and output information.
#nodes: the maximum node number in the circuit.
Node 0(1) is always GND(VDD).
#pn_gnd(vdd) and #nn_gnd(vdd) are the numbers of P- and N-type transistors
     whose drains or sources are connected to GND (VDD).
#gn_gnd(vdd) is the number of transistors that are controlled by GND (VDD).

node_num and node_name are the index and name of the node.
"tg" and "gate" are the indices of TG and gate that the node belongs to.
     If a node is in a trivial TG, both of its tg and gate are 0. If the
     node does not belong to any gate, "gate" is 0.
#pn and #nn are the numbers of P- and N-transistors that are connected
     to the node.
#gn is the number of transistors that are controlled by the node.

<transistor list> is a list of P-transistors, N-transistors connected
     to the node, and the transistors controlled by the node.

#trans: total number of transistors in the circuit.
tran_num and tran_name are the index and name of the transistor.
mos is the type of the transistor. mos = 1(0) for P(N)-type transistor.
tg and gate are the TG and gate that the transistor belongs to.
snode, dnode and gnode are the source, drain and gate nodes of the
```

```
     transistor, respectively.

dir is the signal flow direction of the transistor, which is defined
     as follows, where d and s are the drain and source terminals of the
     transistor, and d-->s denotes that the direction from d to s is
     feasible.

     dir = 0  unknown
           1  d-->s is known, d<--s is unknown
           2  s-->d is known, s<--d is unknown
           3  d<-->s or bidirectional
           4  only d-->s
           5  only s-->d
-------------------------------------------------------------
Example: c60.s.lst

* The node and transistor list file of c60.s
15 PIs: 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
4 POs: 56 58 65 71
The maximum node number is 71
0 GND 0 38 0
1 VDD 49 0 0
2 N2 0 0 0 0 2 1 0
 1 2
3 N3 0 0 0 0 4 1 0
 3 4 5 6
 ...
17 N17 1 1 0 2 0 0 0
 2 4
18 N18 1 1 2 1 2 0 0
 1 3 4 33 34
...
70 N70 32 32 0 2 0 0 0
 108 110
71 N71 32 32 2 1 0 0 1
 107 109 110
110 transistors
 1 MP1 1 1 1 1 18 2 5
 2 MN1 0 1 1 0 17 2 5
 3 MP2 1 1 1 1 18 3 5
 ...
 109 MP55 1 32 32 1 71 69 5
 110 MN55 0 32 32 70 71 69 5
```

Transistor group/gate file

The transistor group file, also called "tg file," usually has the file name "*.tg," where * is the name of the circuit file.

```
File format:

<comment>
#ttgs "trivial TGs and" #tgs "nontrivial TGs"
" Trivial TG nodes" ttg1 ttg2 ...
"TG" TG# tg_type tsn inl outl #trans #nodes #gates #PIs #POs #ins #outs
  "Trans" <transistor list>
  "Nodes" <node list>
  "PI_nodes" <PI list>
  "PO_nodes" <PO list>
  "Inputs" <IN list>
  "Outputs" <OUT list>
"TG" TG# tg_type tsn inl outl #trans #nodes #gates #PIs #POs #ins #outs
  "Trans" <transistor list>
  "Nodes" <node list>
  "PI_nodes" <PI list>
  "PO_nodes" <PO list>
  "Inputs" <IN list>
  "Outputs" <OUT list>
 ...
#gates "GATES"
"GATE" gate# tg# gate_type #ptrans #ntrans #inputs out
  "P-trans" <p-transistor list>
  "N-trans" <n-transistor list>
  "Inputs" <input list>
"GATE" gate# tg# gate_type #ptrans #ntrans #inputs out
  "P-trans" <p-transistor list>
  "N-trans" <n-transistor list>
  "Inputs" <input list>
...
```

```
File description:

#ttgs: number of trivial TGs.
```

```
#tgs: number of non-trivial TGs.
ttg1, ttg2, ... : node indices in the trivial TGs.
TG#: index of the TG.

tg_type: the type of the TG. If the TG is equivalent to a primitive or
     complex gate, tg_type = gate_type, otherwise tg_type = 0.
tsn: topological sorting number of the TG. This information is not used
     in the current system.
inl: input level of the TG, i.e., the number of TGs between this TG and its
     closest PI.
outl: output level of the TG, i.e., the number of TGs between this TG and
     its closest PO.

#trans, #nodes, #gates, #PIs, #POs: The numbers of transistors, nodes,
     gates, primary inputs and primary outputs in this TG.

#ins, #outs: the numbers of inputs and outputs which are not PIs and POs,
     respectively.

<transistor list>, <node list>, <PI list>, <PO list>, <IN list>, <OUT list>:
     The lists of transistors, nodes, PIs, POs, inputs but not PIs, outputs
     but not POs in the TG.

#gates: number of gates in the circuits.
gate#: index for the gate.
tg#: index of TG that the gate belongs to.
gate_type: type of the gate.

     gate_type = 1: inverter
                 2: NAND
                 3: NOR
                 4: Complex gate

#ptrans, #ntrans, #inputs: numbers of P-type transistors, N-type transistors
     and inputs of this gate.
out: the output node of this gate.

<p-transistor list>, <n-transistor list>, <input list>: The lists of
P-type transistors, N-type transistors and input nodes of the gate.
-------------------------------------------------------------
Example: c60.s.tg

*This is the tg file of c60.s.
```

```
15 trivial TGs and 32 nontrivial TGs
 Trivial TG nodes 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
TG 1 2 1 1 4 4 2 1 2 0 0 1
  Trans  1 3 4 2
  Nodes  18 17
  PI_nodes  2 3
  PO_nodes
  Inputs
  Outputs  18
TG 2 2 2 1 4 4 2 1 2 0 0 1
  Trans  5 7 8 6
  Nodes  20 19
  PI_nodes  3 4
  PO_nodes
  Inputs
  Outputs  20
TG 3 2 3 1 2 4 2 1 2 0 0 1
  Trans  9 11 12 10
  Nodes  22 21
  PI_nodes  5 6
  PO_nodes
  Inputs
  Outputs  22
...
TG 32 2 32 3 1 4 2 1 0 1 2 0
  Trans  107 109 110 108
  Nodes  71 70
  PI_nodes
  PO_nodes  71
  Inputs  67 69
  Outputs
32 GATES
GATE 1 1 2 2 2 2 18
  P-trans  1 3
  N-trans  4 2
  Inputs  2 3
GATE 2 2 2 2 2 2 20
  P-trans  5 7
  N-trans  8 6
  Inputs  3 4
GATE 3 3 2 2 2 2 22
  P-trans  9 11
  N-trans  12 10
```

```
  Inputs  5 6
...
GATE 31 31 3 2 2 2 69
  P-trans  105 103
  N-trans  104 106
  Inputs  60 62
GATE 32 32 2 2 2 2 71
  P-trans  107 109
  N-trans  110 108
  Inputs  67 69
-------------------------------------------------------------
```

Fault list file

The fault list file, also called "flt file", usually has the file name "`*.flt`", where "`*`" is the name of the circuit file.

```
-------------------------------------------------------------
File format:

<"BF" node1# node2#> | <"SOP" tran#> | <"SON" tran#> | <"SAN" node# s-a-val>
<"BF" node1# node2#> | <"SOP" tran#> | <"SON" tran#> | <"SAN" node# s-a-val>
...
-------------------------------------------------------------
File description:

Each line starts with either BF, SOP, SON or SAN to represent
     either a bridging, transistor stuck-open, stuck-on,
     or stuck-at fault, respectively.
-------------------------------------------------------------
Example: c60.s.flt

BF 3 10
SOP 25
SON 25
SOP 30
BF 25 30
SAN 35 1
...
-------------------------------------------------------------
```

2.3 I/O specifications for each function

```
g2s:
        Input file: a gate level circuit such as "c60".
        Output file: a circuit file with a spice-like format such as "c60.s"

es2s:
        Input: aldchip.esim
        Output: aldchip.esim.s

dire:
        Input file: a circuit file with a spice-like format such as "c60.s"
        Output on screen: progress of direction assignment, numbers of
                transistors and unidirectional transistors, and
                total process time.

prep:
        Input file: a circuit file with a spice-like format such as "c60.s".
        Output files: a list file such as "c60.s.lst" and a tg file such
                as "c60.s.tg".
        Output on screen: numbers of transistors and unidirectional
                transistors, and total process time.
bfgen:
        Input file: a list file such as "c60.s.lst" generated by prep.
                The program will inquire the circuit file name. The
                correct response for "c60.s.lst" is "c60".

        Input parameters: number of bridging faults and a seed (1-65535)
                to initialize the random number generator.

        Output file: a list of random bridging faults.
                The output file name is obtained by appending ".flt.bf"
                to the input name. For example if the input is "c60"
                then the output is "c60.flt.bf"

asangen:
        Input file: a list file such as "c60.s.lst" generated by prep.
                The program will inquire the circuit file name. The
                correct response for "c60.s.lst" is "c60".

        Output file: all node stuck-at faults in the input circuit.
                The output file name is obtained by appending ".flt.asan"
```

```
            to the input name. For example if the input is "c60"
            then the output is "c60.flt.asan"

sangen:
        Input file: a list file such as "c60.s.lst" generated by prep.
                The program will inquire the circuit file name. The
                correct response for "c60.s.lst" is "c60".

        Output file: all node stuck-at faults in the input circuit.
                The output file name is obtained by appending ".flt.san"
                to the input name. For example if the input is "c60"
                then the output is "c60.flt.san"

songen:
        Input file: a list file such as "c60.s.lst" generated by prep.
                The program will inquire the circuit file name. The
                correct response for "c60.s.lst" is "c60".

        Output file: all transistor stuck-on faults in the input circuit.
                The output file name is obtained by appending ".flt.son"
                to the input name. For example if the input is "c60"
                then the output is "c60.flt.son"

sopgen:
        Input file: a list file such as "c60.s.lst" generated by prep.
                The program will inquire the circuit file name. The
                correct response for "c60.s.lst" is "c60".

        Output file: all transistor stuck-open faults in the input circuit.
                The output file name is obtained by appending ".flt.sop"
                to the input name. For example if the input is "c60"
                then the output is "c60.flt.sop"

switest:
        Input files: the two output files *.lst and *.tg generated by
                prep, and a flt file (*.flt)  containing a list of faults
                to be tested.

        Input parameter: the number of backtrackings allowed for each fault.

        Output on screen: a (pair of) test vector(s) for each detectable
                stuck-at, stuck-on, bridging (stuck-open) fault, and a
                message for each undetectable or aborted fault. It also gives
```

the total test generation time after all faults are processed.

3 Sample run

This section gives an example run using c60. In the example the "%" notation is UNIX system prompt. A line with "---> file_name" denotes that a file with name "file_name" is generated by the command in that line.

```
% g2s c60                                          ---> c60.s
% prep c60.s
Total time for partition and direction assignment is 0.083 seconds.
Total number of transistors =                 110
Total number of unidirectional transistors = 110
start to write lst file ...                       ---> c60.s.lst
start to write tg file...                         ---> c60.s.tg
% asangen
Enter circuit file name: c60                      ---> c60.flt.asan
% bfgen
Enter circuit file name: c60
Enter number of bridging faults: 500
Enter the seed for random number generator: 5     ---> c60.flt.bf
% sangen
Enter circuit file name: c60                      ---> c60.flt.san
% songen
Enter circuit file name: c60                      ---> c60.flt.son
% sopgen
Enter circuit file name: c60                      ---> c60.flt.sop
% cat c60.flt.* > c60.s.flt                       ---> c60.s.flt
% switest c60.s
Enter the limit of backtracks: 100
Node     2 Stuck-at 1 is detected by vector = 0100----------
Node     2 Stuck-at 0 is detected by vector = 1100----------
Node     3 Stuck-at 1 is detected by vector = 10-0----------
Node     3 Stuck-at 0 is detected by vector = 11-0----------
Node     4 Stuck-at 1 is detected by vector = 0100----------
Node     4 Stuck-at 0 is detected by vector = 0110----------
...
BF (    3,   16) is detected by vector = -1------------0
BF (   41,   50) is detected by vector = ------11-011---
```

```
BF (   66,   68) cannot be detected within the limited backtracks.
BF (   43,   45) is detected by vector = 11-0-------0---
BF (   11,   67) is detected by vector = ----0----1-----
BF (   58,   63) is detected by vector = ---11-11-011-1-
...
Node    62 Stuck-at 1 is detected by vector = ----1--0---0-0-
Node    62 Stuck-at 0 is detected by vector = ----1001-011---
Node    64 Stuck-at 1 is detected by vector = ----------11-1-
Node    64 Stuck-at 0 is detected by vector = -----------0--1
Node    67 Stuck-at 1 is undetectable.
Node    67 Stuck-at 0 is detected by vector = ----1--0---0-0-
Node    69 Stuck-at 1 is undetectable.
Node    69 Stuck-at 0 is detected by vector = ----1--0---0-0-
...
Tran     1 SON is detected by vector = 11------------
Tran     2 SON is detected by vector = 01------------
Tran     3 SON is detected by vector = 11------------
Tran     4 SON is detected by vector = 10------------
Tran     5 SON is detected by vector = -11-----------
Tran     6 SON is detected by vector = -01-----------
...
Tran     1 SOP is detected by vectors:
 T1 = 11------------
 T2 = 0100----------
Tran     2 SOP is detected by vectors:
 T1 = -0------------
 T2 = 1100----------
Tran     3 SOP is detected by vectors:
 T1 = 11------------
 T2 = 10-0----------
Tran     4 SOP is detected by vectors:
 T1 = -0------------
 T2 = 1100----------
Tran     5 SOP is detected by vectors:
 T1 = -11-----------
 T2 = -010----------
...
Tran   107 SOP is undetectable.
Tran   108 SOP is detected by vectors:
 T1 = ----0---------
 T2 = ----1--0---0-0-
Tran   109 SOP is undetectable.
Tran   110 SOP is detected by vectors:
 T1 = ----0---------
```

```
 T2 = ----1--0---0-0-
Total test generation time =     3.650 seconds.
% (done)
```

References

[LB88] K.J. Lee and M.A. Breuer. Test Generation System (TGS) user's manual — version 1.0. Technical Report CENG 89-03, University of Southern California, June 1988.

[Lee91] K.J. Lee. *Switch level test generation for CMOS circuits.* PhD thesis, Univ. of Southern California, Los Angeles, August 1991.