14:332:482, 14:332:438, 14:332:480, 16:332:578 Concepts in Digital Systems Design
Instructions for Using the Verilog Automatic Test-Pattern Generator

M. L. Bushnell

March 2, 2005

An *automatic test pattern generator* (ATPG) program takes a logic-level netlist for a Verilog design and generates test patterns for all of the collapsed stuck-at faults in the design. The *spectralatpg* tool uses a novel application of digital signal processing theory, in which all sequential digital circuits are viewed as digital signal processors. The tool stimulates the circuit with random vectors, and compacts the vectors down to the ones that detect faults. The digital spectra of the vector stream are determined, and subsequent vectors are biased to accentuate the spectra shown to be useful for detecting faults. The tool has the highest fault coverages, and the shortest vector sequences, of any existing sequential ATPG tool. The Verilog parser for this tool was developed at the University of California at Berkeley, and given to Rutgers by the Regents of the University of California. The spectral testing group at Rutgers interfaced the parser to the binary netlist data base and the spectralatpg tool.

Figure 1 shows the input file preparation process for the ATPG tool. Figure 2 shows the files read by the ATPG tool, and the files created. All output files, except the **.rub** file, are in ASCII format and can be viewed with a text editor.

1. The **I/P file** is the combined logic-level Verilog description of the design with the **gtech** library description at the beginning.

2. The **.rub file** is a binary data base that has your compiled Verilog design in it. The tool will create this automatically, and will automatically update it whenever you change the Verilog code, so you do not need to worry about this.

3. The **fault_list file** is automatically created by the ATPG tool and has the Verilog signal names of all of the faults in the following format: **from to stuck-at value**, with one fault on every line, with the 3 fields separated by white space. Field **from** is the Verilog name of the signal driving the fault site. Field **to** is 0 if it is not used. For fanout branch faults, **to** indicates the Verilog signal name (load) that the branch fault is on. The **stuck-at value** is 0 for stuck-at-0 and 1 for stuck-at-1.

4. The **vectors file** is the file with test vectors generated by the tool for your circuit. There is a single 0 or 1 for each primary input of your circuit. The order of these signals and the Verilog primary input names for each of these signals is given at the end of the **vectors** file.

5. The **undetected faults file** lists the faults that the tool could not generate tests for. The file format is the same as for the **fault_list file**. You can rerun the ATPG tool on these

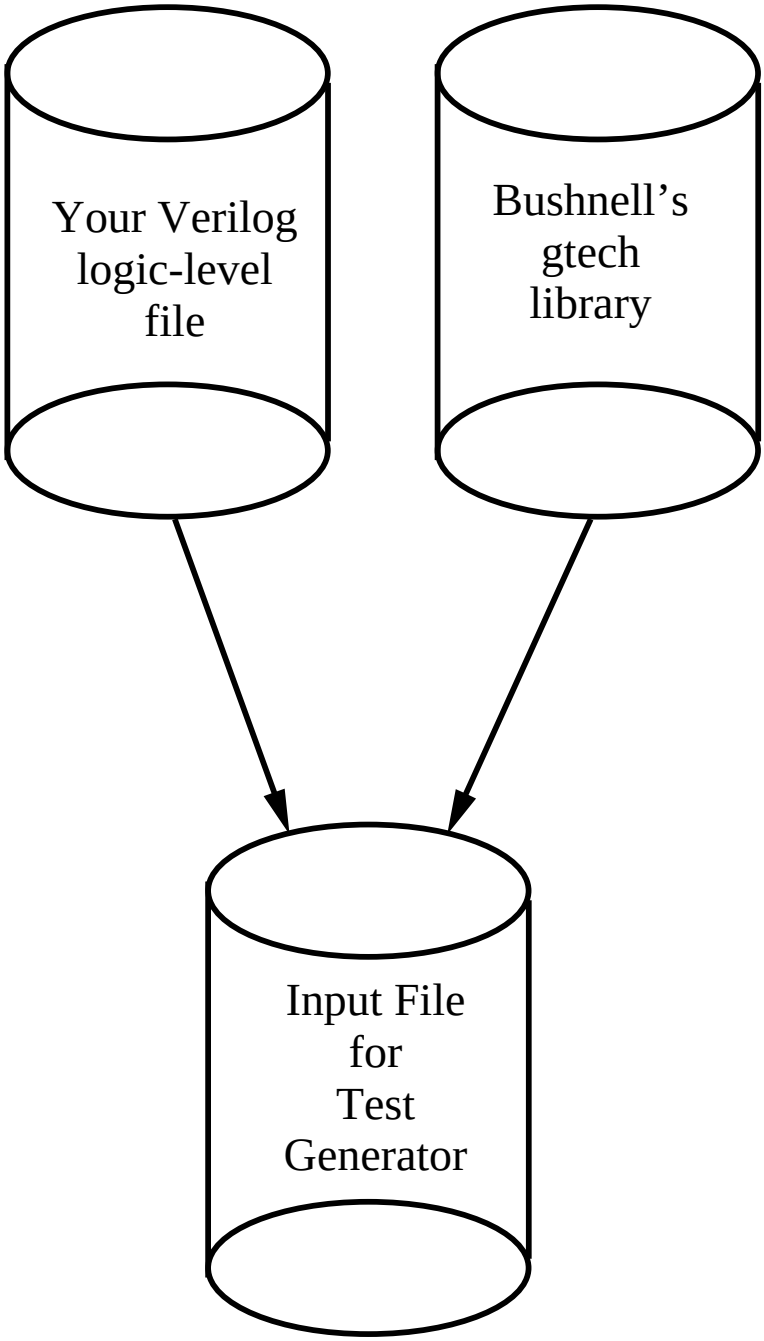

Figure 1: Input File Construction for the spectralatpg Tool.

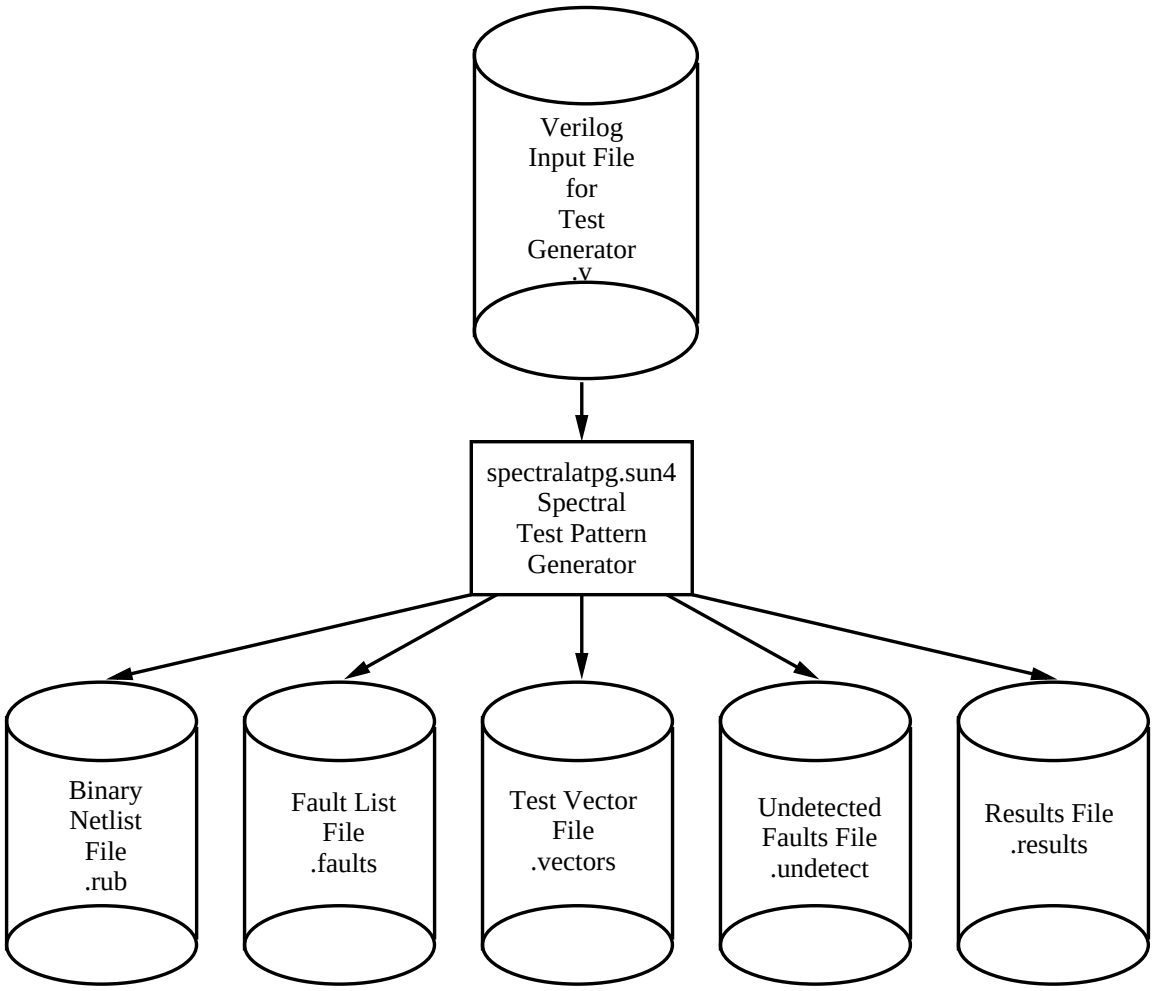

Figure 2: System Data Processing Flow for the spectralatpg Tool.

undetected faults, and it will find more tests for them. You can then place the vectors for the additional faults at the end of the original vectors file. You should not presume that undetectable faults are untestable – you can only conclude that this tool could not find a test for them.

6. The **results file** gives the following important information about the tool's execution:
   (a) For each detected fault, the time frame when it was detected.
   (b) An indication of when the test generator ended a portion of its computation, with the # vectors, the # detected faults, and the fault coverage achieved by that phase.
   (c) The **Final Verification** indicates the final vector length, the total # faults detected, and the final fault coverage. If it is not 100%, you need to add design-for-testability hardware to this design and try again.
   (d) The number of inputs, logic gates and flip-flops, and circuit outputs.
   (e) The # detected faults and # total faults.
   (f) CPU times for portions of the algorithm.
   (g) Memory usage for the algorithm.

   Note that the file does not report *fault efficiency*, because this tool is unable to prove whether faults are untestable.

1 Coding of Verilog for Test Pattern Generation

1. Only logic-level, structural Verilog can be parsed by this tool. This means that if you write behavioral Verilog, you must first synthesize it into logic-level Verilog using the Synopsys **design_analyzer**, and write it out in a logic-level Verilog file from the Synopsys system. Then, you must insert the file ~ **bushnell/atpg/gtech_lib.v** at the beginning of your logic-level verilog file. If you do not insert the library, the tool will not work.

2. The tool will work only with hardware synthesized by Synopsys using the **gtech** library. All bets are off with any other libraries.

3. The name of your topmost module in the Verilog file must be the same as the file name of the logic-level verilog file. Otherwise, the tool will not work for you.

4. The ATPG tool does not understand **parameter**, **'define**, and **assign** statements that assign values to internal wires. If you attempt to use any of these statements, the tool will not work.

5. None of the Verilog transistor primitives or tri-state devices will work with this tool.

6. Any latches synthesized in the output file by Synopsys will cause this tool not to work. However, it does understand all of the various flip-flops used by the Synopsys **gtech** library.

7. If you code the logic-level Verilog file without using Synopsys, you must still satisfy the above constraints. The tool will understand **and**, **nand**, **or**, **nor**, **not**, **xor**, **xnor**, and **buf** primitives in structural Verilog. If you use any flip-flops, you must use only the ones available in the **gtech** library. You can see these in the file ~ **bushnell/atpg/gtech_lib.v**.

8. The tool will reject any Verilog file where internal wires in a module do not drive anything. In fact, this tool will not tolerate any floating wires in the design.

9. If you need to wire signals to logic 1 or 0, you can do this.

10. The parser cannot abide a module with an empty port list.

11. While individual logic gates do not need to have instance names, the parser will unceremoniously core dump if you give a non-primitive module instance without an instance name.

12. The parser cannot interpret the post-2001 Verilog modern syntax. Use the 2001 syntax, instead.

2 Status of the Automatic Test Pattern Generator

The ATPG tool has generated test patterns successfully for a moderate number of Verilog circuits for microprocessors. Here are the known problems:

1. The test pattern generator does not understand module instances with name-value pairs for wiring. Please use only positional wiring to the instance ports.

2. The test pattern generator generates a floating point exception for designs on which it has run for a very long time. We hope to solve this problem shortly.

3 Operating the Automatic Test Pattern Generator

All files for this test-pattern generator reside in the directory ~ **bushnell/atpg** on the caip file server. The tool can be run most simply with this command:

~bushnell/atpg/spectralatpg.sun4 -i input_file_name.v

Use this command to run the tool wherever possible. You can find the advanced options of the tool by typing just **~bushnell/atpg/spectralatpg.sun4**, and it will explain all of its options to you. Here are the options for this tool:

- Basic Usage: **spectralatpg.sun4 [options]**
- Options (can be either upper or lower case):

  -I or **-i <filename>** Name of input file (with **.v** extender).

  -F or **-f filename** Specify the name of the input fault file, when the user wants the tool to generate tests for only certain faults. Otherwise, if this option is not present, the tool automatically generates a complete fault list.

  -J or **-j size** Set the size of the Hadamard matrix to be used (Default:16 for a 16×16 matrix). Must be a power of 2.

  -K or **-k number** Set the maximum number of iterations of the Bit-Perturbation algorithm (default: 30). Must be any positive integer.

  -L or **-l number** Set the maximum number of iterations of the Noisy Matrix algorithm (default: 0). Must be any positive integer.

  -M or **-m number** Set the maximum number of iterations of the Phase-Perturbation algorithm (default: 0). Must be any positive integer.

-N or **-n number** Set the maximum number of iterations of the SG Algorithm 2 (default: 0). Must be any positive integer.

-O or **-o** <**filename**> Specify results file name (defaults to input **filename.results**).

-P or **-p period** Set a fixed period for random Bit-Perturbation (default: in turn taking the following values: 2, 4, 6, 8, 10, 12, 14, 16). Must be an even number.

-Q or **-q cycles** Set a fixed number of phase-shifting cycles for Phase-Perturbation (default: in turn taking the following values: 2, 4, 6, 8, 10, 12, 14, 16).

-S or **-s cycles** Set the maximum holding time of vectors (default: 64). Must be any positive integer.

-T or **-t** <**timeflag**> Time flag to differentiate output file names.

-U or **-u size** Set the size of the BIST Hadamard matrix (Default:8). This must be a power of 2. Prints out the best spectral coefficients for *built-in self-testing* hardware insertion.

-V or **-v cutoff** Set the cutoff value of the BIST Hadamard matrix-filtering operation (Default:4). This must be an integer. Uses this spectral coefficient for calculating the spectra for the **-U** option.

-W or **-w** Produce Verbose output in the results file.

The above four vector perturbation methods are used in the following order:

1. Bit-Perturbation
2. Noisy Matrix
3. Phase-Perturbation
4. SG Algorithm 2

The methods that work best depend on your specific circuit. However, Phase-Perturbation is particularly good for microprocessor and DSP data paths. If the circuit is hard-to-test, increasing the number of iterations for the Bit-Perturbation, Noisy Matrix, and SG Algorithm 2 can be very effective. However, the computational time goes up drastically with more iterations.

4 Help If You Have Problems

We want your feedback, and we will fix either your file or the tool to generate your test patterns. See Omar Khan (Room CORE 609, (732) 445-0844, okhan@caip.rutgers.edu) and describe your problem to him. Have your logic-level Verilog file ready, and be prepared to email it to him.