Maxtestbench
A technology to manage STIL pattern qualification in complex SoC Verification environment

Guillaume Costrel de Corainville / Mickael Broutin

ST Microelectronics
Grenoble, France

www.st.com

ABSTRACT

In STMicroelectronics, the golden pattern reference for exchanges between the design and engineering teams is in STIL format.
To ensure quality and fast test program ramp-up, design needs a solution to ensure pattern behaviour and correctness.

Maxtestbench Synopsys technology is known to address ATPG STIL pattern qualification by simulation. This paper shows how to adapt it to manage "non-ATPG" STIL patterns taking into account the complex SoC verification environment and its constraints.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

Two major categories can be defined in “test world”: functional and structural test.

A structural test solution is mainly based on the fault modelling principle and targets digital circuitry. The Synopsys tools suite is available to address completely the challenge of structural tests by covering

- Structural test implementation into the design using to Design Compiler
- Automatic Test Pattern Generation and fault simulation with the use of TetraMax
- Pattern qualification/validation using the VCS simulator and Maxtestbench verification environment

A functional test solution is mainly based on functionality verification and targets analog/mixed circuitry. Test implementations/concepts are highly dependent on circuit type/complexity. As a consequence, the tool suite and flow to develop functional test patterns differs from a structural test approach.

The aim of this paper is to provide a possible flow convergence path between the two worlds to rationalize tooling and pattern qualification environment in order to reduce the workload without penalizing pattern quality.

2. Design and test overview

The flow described in this paper has been used on a specific design. This chapter aims to provide an overview of the design and structural/functional test covered by this approach.

Design overview

The design project targets the portable gaming market. It has been done in 28nm FD-SOI CMOS technology. A project view in numbers:

- Area of 85mm²
- 4 digital voltage domains
- 3700k DFFs
- 1200 memory instantiations
- Various PHYs (reference list in *Table 2: Functional test category overview*)

Structural test overview

Scan architecture is a core-based methodology. The Synopsys scan insertion tool suite has been used with a partition, topology aware and UPF flow. The scan architecture is provided in: *Table 1: Scan architecture overview*

Mode	DFFs	Compressor-type	Chain length	OCCs
GPU	1400k	7 - sharedIO	200	14
VSS	550k	4 – DFTMax	200	6
CPU	320k	5 - DFTMax	200	7
TOP	1400k	5 - DFTMax	300	168

Table 1: Scan architecture overview

Functional test overview

The functional tests are fully built on an IEEE1149.1 and IEEE1500 access structure. This structure provides 2 kinds of test methodology:

- BIST based methodology
- Direct control/observation methodology (test mode setup, kelvin access setup)

300 functional tests (cf *Table 2: Functional test category overview*) have been developed to cover all memories/PHYs testability aspects.

Functional test	Category	Case Nb	Purpose
Tap and 1500 controller	Direct access	21	Identification, test mechanism access and debug
Boundary scan	Direct access	6	IOs testing and Characterization (except high speed interface)
IO compensation cell	Direct access	4	IDDQ and functional test
Process Monitoring Box	Direct access	12	Adaptive Voltage Scaling purpose and process monitoring
Thermal Sensor	BIST/direct access	1	BIST and parametric test

Memory	BIST	84	Memory BIST
Memory repair	direct sequence	42	Yield improvement
Memory Bitmapping	BIST	3	Memory diagnostic
Memory Retention	BIST	16	Memory power test
PLL	BIST/direct access	30	PLL test, characterization and debug access
DPHY	BIST/direct access	22	calibration, loopback and IOs test and debug access
USB2PHY	BIST/direct access	17	Loopback and various parametric test (resistor value, pull, …)
MiPHY (PCIe)	BIST	26	Loopback and various parametric test (Eye opening marging, beacon test, …)
LPDDRPHY	BIST	9	Loopback and parametric test (delay element, …)
eFUSE	BIST/direct access	7	Fuse test and specific DFT access for read/write operation
RNG	BIST/direct access	4	BIST and debug access
Power Controller	Direct access	2	Specific DFT access/observation to set production test use case.
OCC	Direct access	10	Specific DFT access/observation to set and debug clocking

Table 2: Functional test category overview

3. Pattern generation flow

Initial pattern generation environment

Two different environments/flows have been used. The first environment is managing ATPG patterns and the second one is managing Test Patterns coming from the verification environment. While structural test is fully managed by the DFT team and addressed with a complete tools suite available from CAD vendors, the verification test environment is shared with the verification team and primarily dedicated to verification. The opportunity is taken to reuse verification tests to quickly create silicon (production) tests at the cost of some effort to deal with the constraints of the Automatic Test Equipment (ATE) used, silicon variability and physical constraints. Indeed, a passing test in simulation can fail on silicon and require some rework for many reasons. Typical examples can include:

- Simultaneous transitions which are not achievable on silicon but can exist in simulation
- Glitches that might differ on silicon compared to the modelisation
- Non-synchronized events that can't be generated or observed by the ATE
- Tests generated for a particular process corner (they pass that corner on silicon but might fail on another corners due to a signal shift for example)

Figure 1 Functional test patterns and ATPG verification flows illustrates the two "old style flows".
On the left side, Verification patterns are described using an internal language and applied on the DUT using a simulator running on the DFT simulation platform. DUT IO's are traced and dumped into an EVCD file on which checks are applied to detect basic mistakes, thus ensuring pattern delivery to the engineering team being at the proper quality level.
On the right side, structural test generated with tetramax are directly delivered to the engineering team when the simulation is successful using the Stil2Verilog flow.

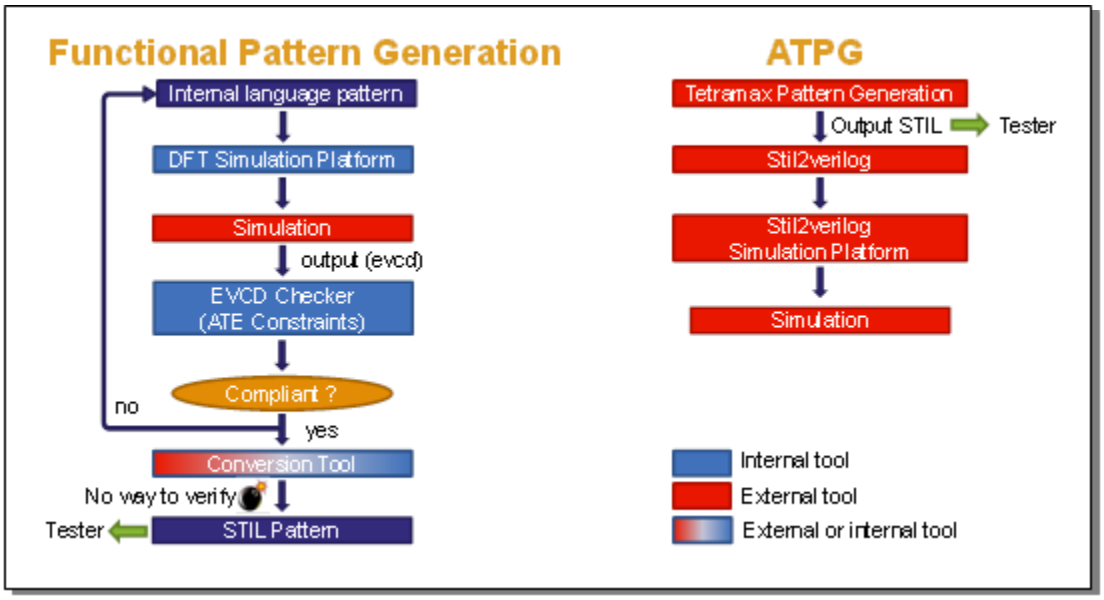

Figure 1 Functional test patterns and ATPG verification flows

As mentioned above, test pattern generation from the verification environment is not straightforward and might require several loops to get robust patterns. Moreover, the usage and maintenance of two flows was perceived as inefficient. The updated environment focuses on potential synergies.

Updated environment

From our initial environment, two flows based on Stil2Verilog capability can be used.
The first is represented with the arrow "What can be done" in *Figure 2 Functional test patterns and ATPG verification flows capitalisation*. This flow allows simulating the STIL pattern resulting from the complete Functional Pattern Generation before using it on the tester. The advantage of this flow is that it can be used in any work environment which would benefit from qualifying any kind of STIL pattern.
The second flow represented with the arrow "What we did" in the following figure is the one actually implemented. The main benefit here is creating the shortest/most efficient path to create and validate STIL patterns. The constraint is that it requires an "internal pattern language" compliant with a cycle based approach and a tool translator into STIL format.

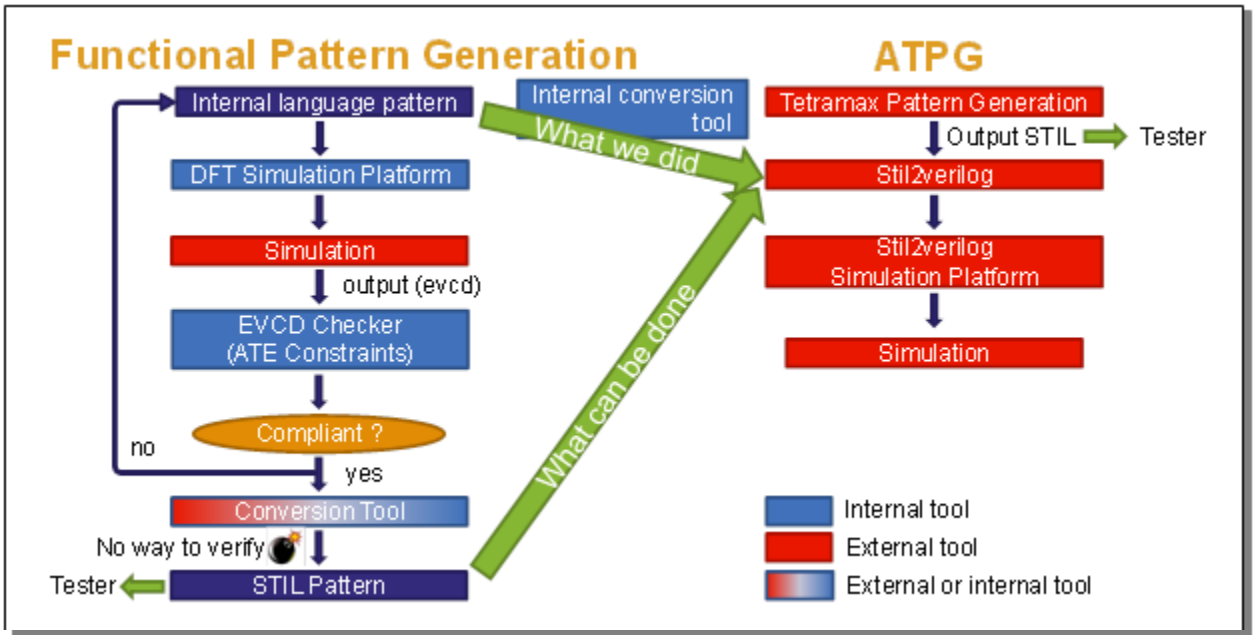

Figure 2 Functional test patterns and ATPG verification flows capitalisation

One additional advantage is the use of a single testbench, thus avoiding specific elaboration per STIL pattern which saves run time and limits the disk space requirements.

On the used design,

- elaboration run time is ~5hours
- elaboration result disk size is ~80Gb
- 200 functional test pattern have been developed

So having a single testbench on the used environment has saved:

- 1000 hours of elaboration runtime (without taking into account pattern debug loop requiring new elaboration)
- 16Tb to allow parallel verification

4. How to generate a generic testbench with Stil2Verilog?

To proceed with respect to the project plan, the Stil2Verilog h-2013.03-sp5 release has been used.

Firstly, the way Stil2Verilog is working is shown in *Figure 3 Stil2Verilog way of working*

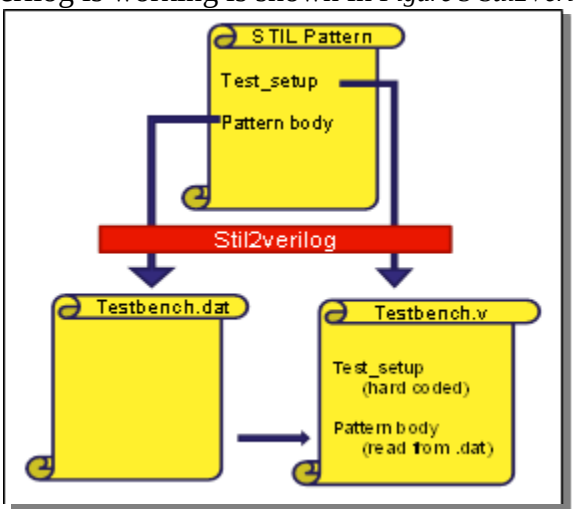

Figure 3 Stil2Verilog way of working

Stil2Verilog reads a STIL pattern and creates a verilog testbench and ".dat" sequence file.
The Testbench is mainly a stimuli/checker generator based on a ".dat" file directive. The ".dat" file directive describes all input/output activity.
By default, the created testbench contains some pattern dependant information such as WaveForm tables, Test setup, loops, … which require dedicated testbenches per pattern.
The following part focuses on how to use the MaxTestbench/Stil2Verilog flow to manage functional test patterns and how to create a generic/unique testbench environment whatever the functional test pattern.

Stil2Verilog Flow to manage functional test pattern

The command line used to run Maxtestbench is:
>> Stil2Verilog pattern_name.stil testbench –replace
In this command line, pattern_name.stil is the pattern for which the testbench needs to be created, and testbench is the name of the generated files (testbench.v for the verilog testbench and testbench.dat for the test data file).

To be able to run Maxtestbench on a STIL pattern to generate a testbench, it is necessary to have a header containing the name of the design under test.
In our case we used: Title "Minimal STIL for design `chip_top'";
This information is used by Maxtestbench to instanciate the design under test in the testbench.
If this is not present, Maxtestbench will not proceed and will report the error
maxtb> Error: Cannot retrieve DUT module name in STIL file. Set the "cfg_dut_module_name" in the config file to avoid the problem. (E-035)

Once this header is properly set, two approaches are possible:

1. The first is to use an ATPG pattern to reuse its structure by replacing the test_setup part by the functional test pattern sequence. The problem faced with this approach is that the test_setup is hard coded in the verilog testbench generated by Maxtestbench. A generic testbench generation is not possible due to a test setup which differs for each functional test pattern (test mode configuration + test muxing configuration + clocking configuration)
2. The second approach is to generate a STIL pattern with an empty test_setup, and to put the whole sequence (dedicated chip setup + test sequence) into the "pattern" part of the generated STIL. Then, all the testbenches (corresponding to different testcases) are generated without the test_setup and the complete pattern is read from an external file. Proceeding like this, the generated testbench become almost generic.

Only the second approach has been used to fully qualify functional test patterns. It is the most efficient solution in term of environment re-use, disk space, and run time compared to the first approach.

Based on the test suite used, some flow updates have been done to ensure testbench genericity.

STIL2Verilog flow update for Testbench genericity

Update 1: Loops present in a pattern, either in the „test setup" or „pattern" parts, are hard coded in the generated testbenches
Impact: the testbench is pattern dependent (ie. Loop information is coded in the "testbench.v" instead of the ".dat" file)
Solution: A script is run on the STIL pattern to replace the loops by the corresponding number of iterations of the vector looped.

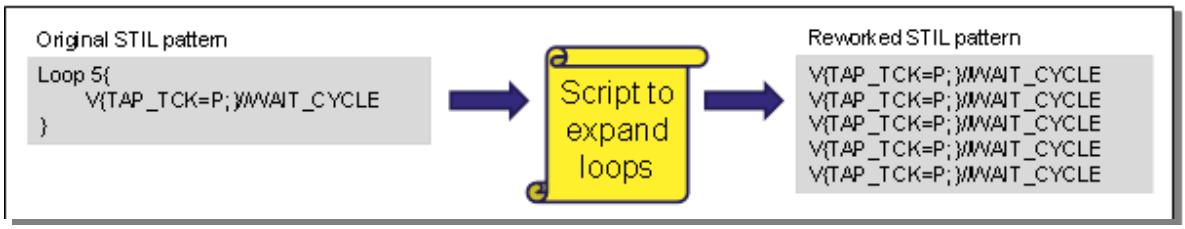

Figure 4 Script to expand loops

Update 2: In some, but not all, testcases, a check on outputs is done using _PO group
Impact: the testbench is pattern dependent (ie. the generated testbenches have not the same signal list)
Solution: To generate a common signal list in all testcase testbenches, a dummy test on _PO is added at the end of each testcase.

Figure 5 Dummy check

Update 3: Two registers called memel and memall are hard coded in the testbench, and are different for each testcase's testbench.
Impact: The testbench is pattern dependent (ie. Parameter dependency of the ".dat" file content)
Solution: A script is used to get the sizes of the memel and memall registers and to store them as parameters in a local file for each testcase simulation directory. The final generic testbench is modified to replace these registers' sizes with the stored parameters, then at simulation time the values stored localy for each testcase are used to overwrite the parameter value in the generic testbench.

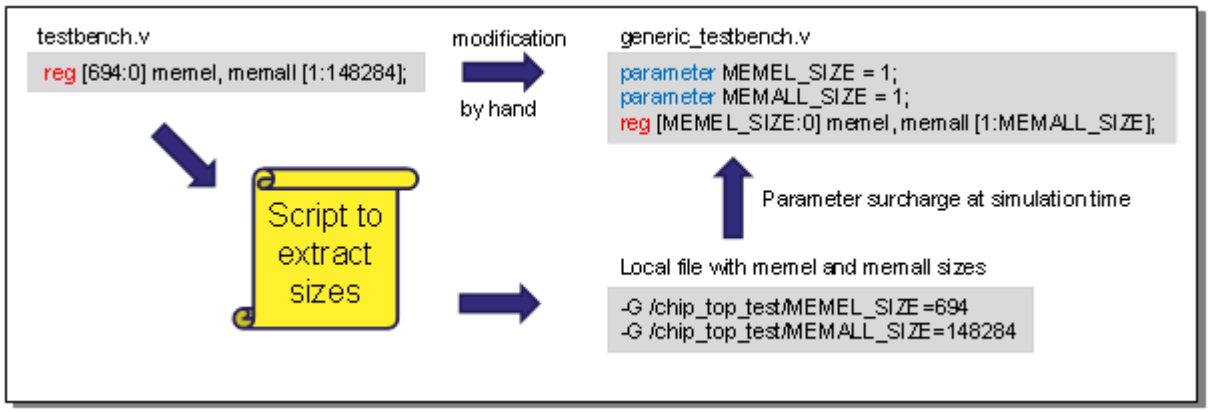

Figure 6 Memel/memall genericisation

Update 4: The TDATA file path is absolute
Impact: It is impossible to run several simulations in parallel
Solution: Update the TDATA FILE path from an absolute to a relative path to be able to launch all the tests generated with same names in different simulation directories. In our case, the update is `define TDATA_FILE "../testbench.dat"

Update 5: IOs management
Impact: the testbench is pattern dependent and IOs management is not correct
Solution: For all the inout PADs, the task apply_XXX_default_WFT_WFT must be updated. In the following figure, the two blue parts show the updates done.

```
task apply_<YOUR_NAME>_default_\WFT_\WFT;
  reg [SIG_IDS_\W-1:0] sid;
  reg [SIG_IDS_\W-1:0] n;
  reg [SIG_IDS-1:0] s;
 begin
  for (sid=0; sid < SIG_IDS; sid=sid+1) begin
   n=0;
   case(sid)
    'd1, 'd2, 'd3, ..., GPIOxx
    begin
     if(ALLOUTSIGIDS[sid]==1'b1) begin
      for (n=0; n <SignalIDWidth[sid]; n=n+1) begin
       case ({TMPOUTSIGS_T[MAX_SIG\W*sid+n],TMPOUTSIGS_\V[MAX_SIG\W*sid+n]})
        2'b10: begin   ALLINSIGS[MAX_SIG\W*sid+n] <= #(0) 1'bZ;   ALLOUTSIGS[MAX_SIG\W*sid+n] <= #(0) 1'bX;   ALLOUTSIGS[MAX_SIG\W*sid+n] <= #(30) 1'b0;  end
        2'b11: begin   ALLINSIGS[MAX_SIG\W*sid+n] <= #(0) 1'bZ;   ALLOUTSIGS[MAX_SIG\W*sid+n] <= #(0) 1'bX;   ALLOUTSIGS[MAX_SIG\W*sid+n] <= #(30) 1'b1;  end
        2'b1X: begin   ALLINSIGS[MAX_SIG\W*sid+n] <= #(0) 1'bZ;   ALLOUTSIGS[MAX_SIG\W*sid+n] <= #(0) 1'bX;   ALLOUTSIGS[MAX_SIG\W*sid+n] <= #(30) 1'bX;  end
        default: undef_wfc({TMPOUTSIGS_T[MAX_SIG\W*sid+n],TMPOUTSIGS_\V[MAX_SIG\W*sid+n]}, sid, 0);
       endcase
      end
     end
     if(ALLINSIGIDS[sid]==1'b1) begin
      for (n=0; n <SignalIDWidth[sid]; n=n+1) begin
       case ({TMPINSIGS_T[MAX_SIG\W*sid+n],TMPINSIGS_\V[MAX_SIG\W*sid+n]})
        2'b00: begin   ALLINSIGS[MAX_SIG\W*sid+n] <=   #(0) 1'b0;  end
        2'b01: begin   ALLINSIGS[MAX_SIG\W*sid+n] <=   #(0) 1'b1;  end
        2'b0Z: begin   ALLINSIGS[MAX_SIG\W*sid+n] <=   #(0) 1'bZ;  end
        2'bZ1: begin   ALLINSIGS[MAX_SIG\W*sid+n] <=   #(0) 1'b0;  ALLINSIGS[MAX_SIG\W*sid+n] <= #(33) 1'b1;  ALLINSIGS[MAX_SIG\W*sid+n] <= #(66) 1'b0;  end
        2'bZ0: begin   ALLINSIGS[MAX_SIG\W*sid+n] <=   #(0) 1'b0;  end
        default: undef_wfc({TMPINSIGS_T[MAX_SIG\W*sid+n],TMPINSIGS_\V[MAX_SIG\W*sid+n]}, sid, 0);
       endcase
      end
     end
    end
```

Figure 7 Task apply_XXX_default_WFT_WFT modification

The first update is necessary to have all the inputs signals in high Z instead of any value that could create a multiple drive and a measurement issue.
The second update is needed to ensure Verilog testbench and verification environment unicity, whatever the pattern.

5. Achievement and Conclusions

Currently, Maxtestbench is not intended to cover functional test pattern qualification problems but it is possible.

The presented solution has been implemented and proven up to silicon and ATE validation.
On the current project, 200 functional STIL patterns have been validated with 1 man week's work effort using this solution.
On the first day of silicon availabilty, all test access and all memory BISTs have been validated on the ATE. This pattern flow has demonstrated a robust pattern development and eased test program ramp-up (cf *Figure 8: Pattern validation progress on ATE*).

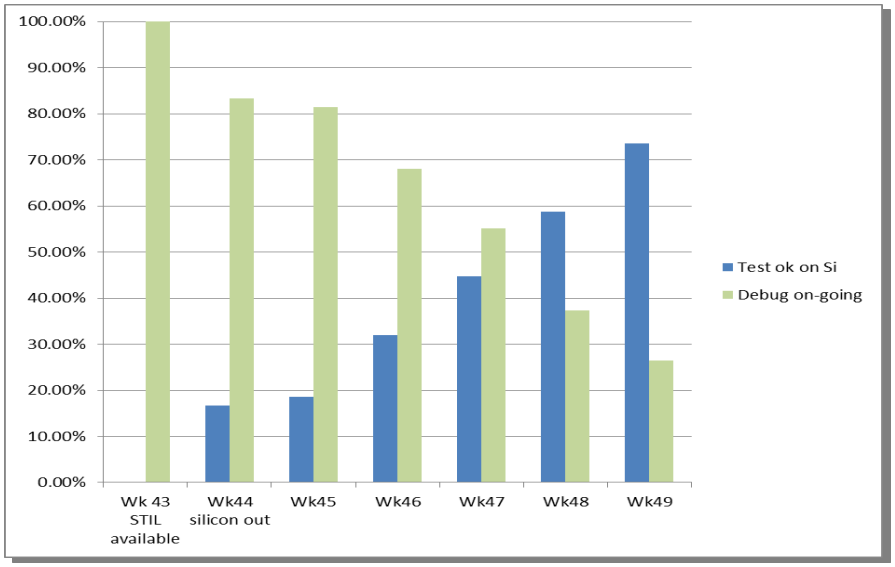

Figure 8: Pattern validation progress on ATE

This approach provides the following benefits:

- Reduces test program ramp-up and resource debug effort by providing a Synopsys solution to qualified functional STIL pattern.
- Enables flow rationalization by reducing the number of steps and tools required for functional test pattern generation.
- Creates flow synergy between the functional and structural test worlds.

This paper reports some limitations to efficiently manage functional test pattern qualification. Most of these limitations are already obsolete thanks to the latest Maxtestbench version (loop management, Verilog testbenches have no more test setup dependency)

The proposed flow could be used to create synergy inside the SNPS tool suite. SMS and SHS Synopsys technologies are targeting Memory and IPs STIL pattern generation. Maxtestbench coming from the TetraMax tool suite could be used to validate these patterns.

In the proposed flow, one step is still based on an internal solution: “internal pattern language”. It limits the efficient pattern reuse inside groups/divisions of STMicroelectronics and the exchange of data with external providers and customers. The next problem is standardization of this step. The IJTAG standard could provide an answer by proposing a Procedural Description Language.

6. References

Ref	Title	link
[1]	MaxTestbench User Guide	MaxTestBench User Guide link
[2]	IEEE Standard Test Access Port and Boundary-Scan Architecture	IEEE Std 1149.1-1900
[3]	IEEE Standard Testability Method for Embedded Core-Based Integrated Circuits	IEEE Std 1500-2005
[4]	IEEE Standard for Access and Control of Instrumentation Embedded within a Semiconductor Device	IEEE Std 1687-2014

Table 3: Reference table