Fault: Open-Source EDA's Missing DFT Toolchain

Manar Abdelatty, Mohamed Gaber, and Mohamed Shalan
The American University in Cairo (AUC)

Editor's notes:
An open-source DFT flow is essential for any open-source solution. This article describes an approach to fill in this missing piece.
—Sherief Reda, Brown University
—Leon Stock, IBM
—Pierre-Emmanuel Gaillardon, University of Utah

However foolproof current EDA tools happen to be, fabricated circuits may not function correctly because the manufacturing process itself is imperfect. Defects, such as short circuits and open circuits, may be introduced because of these imperfections. Therefore, manufactured circuits are typically tested against defects before packaging, as it is crucial to identify faulty circuits as early as possible; when the faulty chip is soldered on a printed circuit board, the cost of fault remedy would be multiplied by ten. This cost factor continues to apply at every step until the system that uses the chip is delivered to the end user, referred to in the industry as "the rule of ten."

Regardless the wide variety of proprietary and commercial design-for-testability (DFT) toolchains available, there is a surprising dearth of open-source DFT toolchains. There are some freely available tools with limited utility, such as the Atalanta [1] automatic test pattern generator, which, even then, is not open source as it has a number of usage restrictions. In addition to the source code being freely customizable for research purposes, the reality is that the primary purpose of open-source software is to not only break down the barriers of entry to EDA development but also provide a publicly available tool for research and study. Also, while the proprietary tools are typically available at a nominal cost to educational institutions, in the industry, they command outrageous, cost-prohibitive prices to startups; erecting barriers in the EDA space and inhibiting innovation in the field. Fortunately, the barrier to fabricating the actual chips is slowly being broken down by projects like OpenROAD and Google Open-Process Design Kit (PDK), which necessitates the existence of an open-source automated flow for DFT.

Testing of digital logic circuits involves the application of test data (test pattern/vector) to the device under-test (DUT) and the comparison of the resulting response to an expected one as produced by a known-good model [the golden model (GM)]. Should a manufacturing defect be able to alter the behavior of a circuit, a discrepancy would appear between the DUT and the GM, which allows for the DUT to be quickly pointed out as a defective chip. At the core of this process is what is known as test pattern generation, which aims to find a set of input sequences that would be able to detect faulty circuits.

Test pattern generation is a complex process with two primary aspects to optimize: 1) the cost of test application (proportional to the testing time) and 2) the quality of the tests (coverage). In essence, automatic test pattern generation (ATPG) software is

Digital Object Identifier 10.1109/MDAT.2021.3051850
Date of publication: 14 January 2021; date of current version: 8 April 2021.

designed to minimize the number of generated test vectors (TVs) (and therefore, lessen the amount of time spent in testing) while maximizing the number of covered fault sites to ensure that in that testing, as many defects as possible are covered. A less important but still significant nonrecurring engineering cost is the time spent on generating the ATPGs: for larger, more complex circuits especially with millions, if not billions of fault sites, the ATPGs should also aim to minimize the amount of time taken to generate the TV set.

DFT tools also typically include the infrastructure required for a circuit to support such testing: because the many possible issues that may arise during the fabrication of a hardware design require almost any hardware written to be designed with testability in mind. To this extent, standards have been introduced to assist with automated testing, most famously IEEE 1149.1 [2], which is commonly known as Joint Test Access Group (JTAG).

The EDA research team at the American University in Cairo has thus endeavored to create such a tool: one that leverages existing open-source tools such as the Yosys Open SYnthesis Suite [3], the Icarus Verilog simulator [4], and the Pyverilog [5] to deliver a cohesive experience encompassing netlist cutting, ATPG, static compaction all the way to scan chain insertion, JTAG interface stitching, and verification. We succinctly call this toolchain "Fault." Fault is designed and implemented to support standard EDA formats; hence, it can be integrated into any industrial RTL to graphic design system II (GDSII) flow.

Design-for-testing overview

Fault models

Because of the diversity of VLSI defects, it is difficult to generate tests for real defects. Fault models are necessary for generating and evaluating a set of TVs. *Fault modeling* is a process by which possible fault sites can be represented and simulated behaviorally, regardless the actual cause. Also, it should be computationally efficient in terms of fault simulation and test pattern generation. Many fault models have been proposed, but, unfortunately, no single fault model accurately reflects the behavior of all possible defects that can occur. However, the single stuck-at fault model is widely used and considered the *de facto* standard fault model as it has been used successfully for decades.

Stuck-at-faults

A stuck-at fault affects the state of logic signals on connections in a logic circuit, including primary inputs (PIs), primary outputs (POs), internal gate inputs and outputs, fanout stems (sources), and fanout branches. We refer to them as *fault sites*. Generally, the number of fault sites is equal to the sum of number of PIs, the number of gates, and the number of fanout branches. A stuck-at fault transforms the correct value on the faulty signal line to appear to be stuck at a constant logic value, either a logic 0 or a logic 1, referred to as stuck-at-0 (SA0) or stuck-at-1 (SA1), respectively.

Delay faults

A delay fault is described as a fault that is not inherently structural, rather, it is a fault that prevents the device from operating at the desire clock speed. While chips with structural imperfections are obviously undesirable, chips that cannot run at the specified clocking requirements are also functionally useless, that is, outside of a limited class of general-purpose computing hardware that does not need determinism, making stuck-at faults not completely sufficient for circuit testing. Of particular interest is the so-called transition fault model, which is easy to implement by modifying stuck-at ATPG software [6].

Combinatorial equivalence

As the vast majority of VLSI circuits are sequential, the stuck-at faults are, by nature, limited as a register would typically lie between an input and an output, making it so a stuck-at fault would typically not propagate from an input to output, therefore, making tests useless. Therefore, a method must be utilized to somehow bypass the registers for the purposes of TV generation and testing. There are two methods used: one while generating TVs, which is *register cutting*, and another for fabricating the actual chip, which is the use of a scan chain.

Cutting

"Cutting" the circuit refers to the act of replacing every D-flipflop with an input and an output so they can be easily accessed while generating test patterns. As shown in Figure 1, each flip-flop's output becomes an input for the rest of the circuit and each flip-flop's input becomes an output for the rest of the circuit. In that manner, it is possible to test all inter register logic by manipulating the register outputs as desired and evaluating the register inputs alongside testing regular inputs and outputs. The circuit

generated by this process is ephemeral; however, it is only used for pattern generation and then discarded.

Scan chain generation

To utilize these test patterns, which also include the registers with actual circuits, the circuits must include some manner of infrastructure to allow the manipulation of register values: both writing the "inputs" and reading the "outputs." This is typically approached by connecting *all the registers in a circuit* in a process known as scan chain stitching, where every register is serially connected to another register in the circuit forming a full serial chain. A test-mode pin is usually added to multiplex between the regular interregister logic and scan chain logic. The scan chain is self-verifying: simply scanning a pattern in and out fully is sufficient to ensure that not only the scan chain is functional but also ensure that all registers in the circuit are functional as well: failing to scan a pattern in and out is in itself a quick way to realize that a circuit is faulty.

Automatic test pattern generation

Test generation is the task of producing an effective set of TVs that will achieve high fault coverage for a specified fault model. In general, ATPG tools differ by the used fault model and the approach or algorithm used for the TVs generation approach.

The other key characteristic of ATPG is the method used to generate the TVs. Random test generation (RTG) is the simplest method for generating TVs. TVs are randomly generated and fault-simulated (simulated in the presence of faults) till a reasonable high fault coverage by the TVs is achieved.

There are also a number of ATPG algorithmic methods that are in wide use today, including the D-algorithm, the path-oriented decision making (PODEM), and the fan-out oriented algorithm (FAN) [7]. Pattern generation through any of these algorithmic methods requires what is known as path sensitization; which refers to fault activation and propagation by finding a path in the circuit that will allow a fault to show up and propagate to an observable output of the circuit if it is faulty.

For the example circuit shown in Figure 2, the ATPG process identifies eight fault sites {A, B, C, d, e, f, g, Y}, and 16 stuck-at faults. To simulate stuck-at faults, the ATPG constructs a faulty model for each generated TV by forcing each fault site to be stuck-at zero and stuck-at one. Then, the output of the faulty model is compared to the GM output

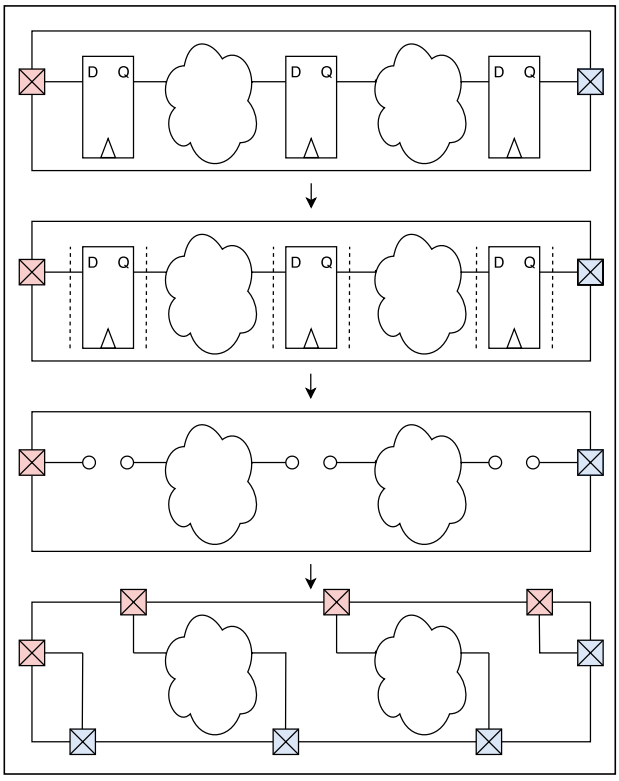

Figure 1. Converting a sequential circuit to a combinational circuit by "cutting." The inputs are marked in red and outputs are in blue.

and discrepancies mark a fault as detectable by the applied TV. For example, applying the input sequence {1,0,1}, while forcing node e to be stuck-at one, would produce an incorrect circuit result which means e's stuck-at one fault is detectable by the applied TV. However, forcing e to be stuck-at zero would result in a correct circuit output meaning that e's stuck at zero fault is not detectable by the applied vector. This process is repeated for all sites, for each generated vector, and then coverage is computed.

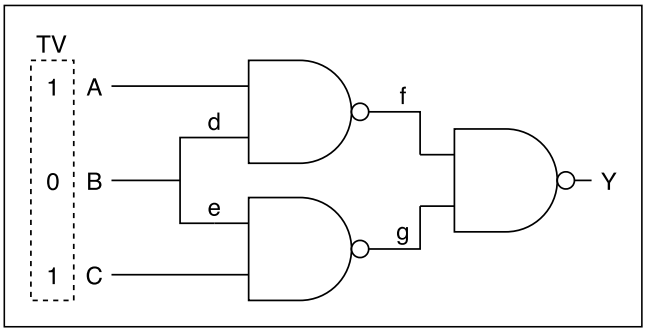

Figure 2. ATPG example. The fault sites are lettered.

Fault toolchain

Fault operates on synthesized netlists in Verilog and is made up of five components: Cut, PGen, Compact, Chain, and Tap. Figure 3 shows the typical design flow of Fault. First, the flattened netlist is converted into a pure combinational design using Cut. This modified netlist is used for the ATPG process done by PGen which outputs the final coverage and the generated TVs in a javascript object notation (JSON) format. The generated TV set is then compacted by Compact that reduces the number of the generated TVs without affecting the coverage. Finally, scan chain insertion is done by Chain and a JTAG controller is stitched to the inserted scan chain by Tap.

Cut

Using Pyverilog [5], the flattened netlist flip-flops are removed converting the sequential design into a pure combinational design. The new netlist has an extra input port for every removed flip-flop output pin and an extra output port for every removed flip-flop input pin.

PGen

PGen is used to perform ATPG for stuck-at faults. The stuck-at fault model assumes that manufacturing defects cause nodes to be stuck at logic zero or logic one. PGen uses pseudorandom ATPG coupled with fault simulation. This is a simpler alternative to algorithmic methods such as PODEM and D algorithms. Algorithmic methods require "path sensitization," which makes them complex to handle netlists mapped using any arbitrary standard cell library. In PGen, TVs are pseudorandomly generated then simulated. PGen generates a testbench, for every generated TV, that compares a GM to a model where fault sites are progressively injected using Verilog force statements. The outputs of both models are compared, and any fault site that can be marked as detectable using said TV is sent back to Fault to be marked as covered. PGen stops generating TVs when either the target coverage or the maximum number of TVs has been reached. Final coverage is then output to a file in a ubiquitous and easy to manipulate JSON format.

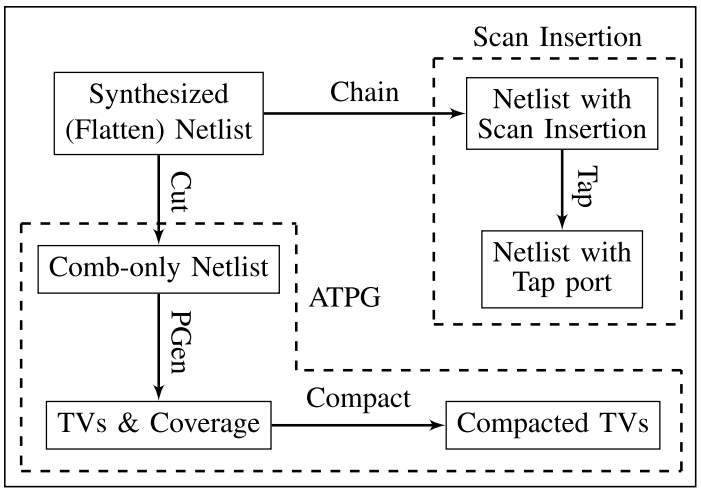

Figure 3. Fault design flow.

Compact

Reduces the size of the TV set using static compaction while maintaining the coverage percentage of the initial test set. The compaction is needed to reduce the testing time; hence, reducing the cost. The compaction process is illustrated in Algorithm 1. It starts with two sets: the initial TV set, generated by the ATPG, along with its covered faults and an empty compacted TV set. First, essential TVs (i.e., ones that cover at least one fault not covered by any other TV in the set) are added to the compacted set unadulterated, while the initial test set has the essential fault points removed. Then, the TV with the highest number of detectable faults is inserted in the compacted set and the faults covered by that vector are removed from the initial test set. This process is repeated until the compacted set covers all previously detected faults.

Chain

Performs scan chain insertion. Chain converts a netlist's flip-flop cells to scan cells by adding an abstract multiplexer definition to every flip-flop input port using Pyverilog [5]. This definition is then mapped by Yosys [3] to either a multiplexer (MUX) cell from the standard cell library or a scannable

Algorithm 1: Static TV Compaction.

```
Data: S1 – TV set
      F1 – Detectable Faults
Result: S2 – Compacted TV set
initialize S2 ;
add essential TVs to S2;
while S2 does not cover all faults in F1 do
    Find TV that covers the highest number of faults;
    add TV to S2;
    Remove faults covered by TV from F1;
end
```

flip-flop cell if the library has one. The chain also adds boundary scan cells for the netlist's inputs and outputs, then stitches all the scan cells into one scan chain. The chained netlist has extra ports for the serial-in and serial-out of the constructed scan chain in addition to the testing control signals. Additionally, Chain offers an option to automatically generate a testbench to verify the scan chain integrity.

Tap

Tap adds the JTAG interface to a chained netlist using Pyverilog. This is accomplished by adding its five namesake test access ports (TAPs): serial test data in (TDI), test mode select (TMS), serial test data out (TDO), test clock (TCK), and active low test reset (TRST). As illustrated in Figure 4 the serial-in and serial-out of the constructed scan chain are stitched to the TDI line and the input of the TDO line multiplexer respectively. Selecting a register between TDI and TDO lines is done by shifting an instruction code on the TDI line. For the purpose of doing on-chip testing, a custom instruction is defined to select the internal scan chain. The TAP controller is verified by automatically generating a testbench wherein TVs are shifted through the scan chain, then applied to the onchip logic by deasserting shift for one clock cycle, then the captured output is shifted out and compared to the fault-free response.

Implementation

The fault is implemented in the Swift programming language [8] as it is a statically typed, safe, native programming language that could also interact with and use Python-based libraries idiomatically.

Fault leverages the Swift–Python interoperability developed by Google Inc. as part of their Swift for Tensorflow project [9], allowing it to interface seamlessly with the Pyverilog library which produces an abstract syntax tree for direct manipulation, necessary for the cutting behavior, scan chain stitching, and other things. Most logic is implemented in pure. Swift, which, while compiled Python yields a negligible speed boost, the native programming language is more lenient on memory usage, which is a great boon when simulating large hardware designs.

Interfacing with Yosys and Icarus Verilog is done via simple calls to the UNIX shell: Fault would generate the synthesis script or testbench, respectively, call the tool responsible and parses its output, which may be written to either a pipe or a file, which is then read by

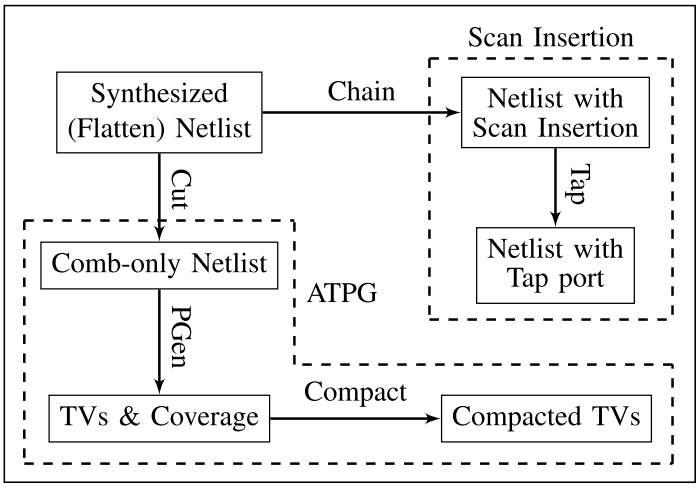

Figure 4. JTAG controller connection to a chained netlist.

Fault. The fault runs many simulations in parallel and benefits greatly from a multithreaded environment.

Despite that, however, a practical problem is that setting up the Swift language, let alone Swift/Python interop, is cumbersome on Linux, which is a great concern considering cloud infrastructure overwhelmingly runs Linux—not to mention that the problem becomes greater while distributing for users as Windows support for Swift is immature and Apple does not yet provide official binaries for the language on Windows. To help alleviate these problems, lightweight Docker images were created so one may run Fault on any platform of their choice in a reliable, relatively configuration-free setting.

Benchmarking and performance

We have evaluated the performance of Fault's flow on a number of open-source designs frequently used as benchmarks. The coverage and runtime are used as metrics for the ATPG process. Coverage results were obtained by running the ATPG process with a ceiling of 5000 TVs, an increment of ten TVs per iteration, and minimum coverage of 97% such that the ATPG process stops when the ceiling count is encountered or the minimum coverage is achieved. As shown in Figure 5a, the fault is able to achieve 96.6% coverage on average. As shown in Figure 5b, Fault's runtime is moderately low for smaller designs, but it is significantly higher for larger designs. This is largely attributed to the ATPG problem being NP-complete [10] thus needing to generate a larger TV set to reach a reasonable coverage;

hence, more Iverilog simulations. The fault simulations were carried out on Intel Xeon-based station running Ubuntu 18.04 long-term support (LTS) with Fault's native installation with an allocated RAM of 32 GB and ten threads.

Additionally, we experimented with Atalanta to validate Fault's stuck-at simulator. The experiment involved generating the TVs with Atalanta, then simulating the generated vectors using Fault's stuck-at simulator. Since Atalanta does not support standard EDA formats and is only compatible with a netlist in the International Symposium of Circuits and Systems (ISCAS) bench format, we introduced an optional utility to Fault's flow, bench, that takes a gate-level netlist and the standard cell library Verilog models as an input and generates the netlist in bench format. This netlist was then used as an input to Atalanta. Figure 5c shows the coverage results for Fault's simulator and Atalanta. Fault's coverage is slightly lower than Atalanta; however, Fault shows an accurate coverage percentage of the gate-level netlist because the bench circuit format does not support some circuit constructs such as being permanently grounded.

For the compaction process, the metric used is the size of the initial TV set versus the compacted set. As shown in Figure 5d, Fault is able to significantly reduce large TV sets (5000) with a reduction percent of 97.2% on average. For smaller sets, the reduction percentage is lower because most of the generated TVs are essential. The compaction process was also verified by rerunning the fault simulations with the compacted set to ensure that the coverage percentage is not reduced.

For the scan chain insertion and JTAG controller stitching, the area overhead is calculated to evaluate the penalty of DFT. As shown in Figure 5e, for smaller designs, the area overhead is quite large (i.e., >90%) rendering it unnecessary, however, for larger designs, the cost of adding extra testability logic is insignificant making the strong case of the necessity of having DFT structures for complex designs especially.

Using fault for ASIC tape-out

With the advent of open-source EDA movements, the release of Google SkyWater PDK, and openLane project, we are able to work on a potential tape-out of striVe6 chip [11], a PicoRV32 SoC with testability structure automatically injected by Fault. The fault was able to achieve a coverage percentage of 91% on striVe6 with 5000 TVs that were compacted to 311 TVs and verified to have the same coverage as the original set. The scan chain and JTAG controller were automatically constructed and verified by Fault with an estimated area overhead of 16%.

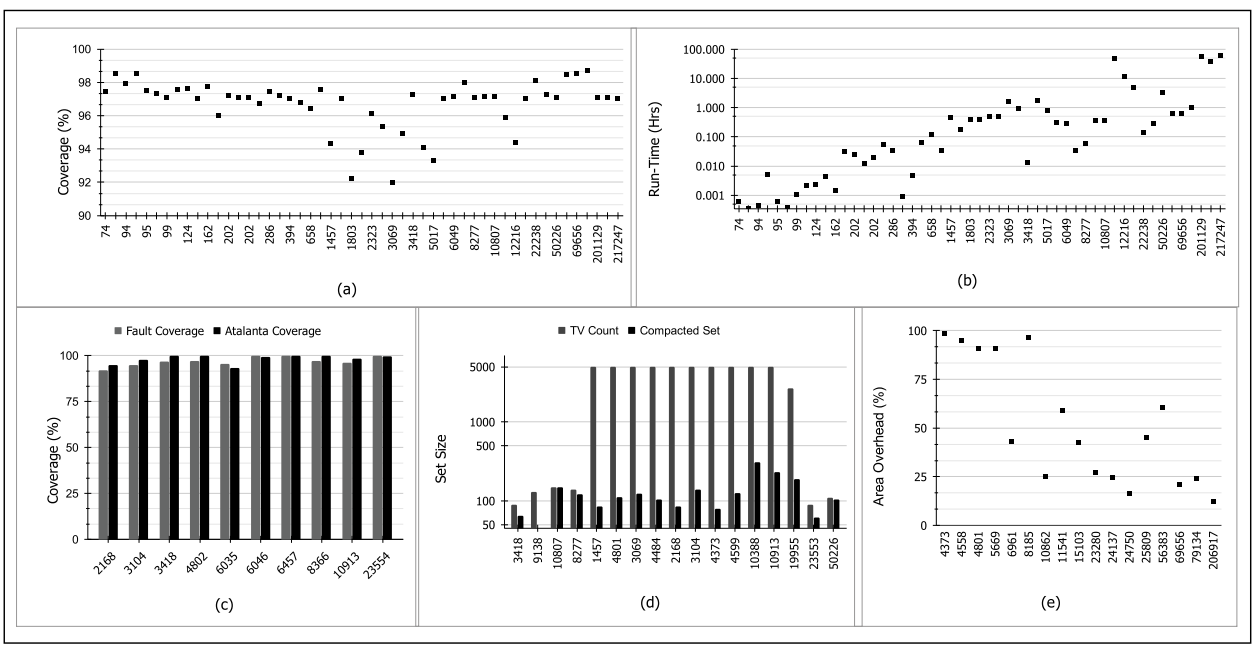

Figure 5. Fault's flow results. (a) Coverage versus gate count. (b) Run-time versus gate count. (c) Fault and Atalanta coverage versus gate count. (d) Initial TV set versus compacted set count. (e) Area overhead versus gate count.

The striVe6 exercise has revealed several challenges with Fault's flow. First of which is how to deal with unscannable black-box modules (like SRAM) in the ATPG process. This was solved by extending the Cut option to support removing black-box modules and, like flip-flop cells, exposing the blackbox module PIs as output ports and POs as input ports; thus bypassing the SRAM block while testing. Additionally, the Chain option was extended to support bypassing black-box modules. This was achieved by adding a wrapper scan cell to the black-box PIs and POs that were eventually stitched to the flip-flops scan chain. The second challenge was having flipflops with different clock-edge sensitivity, which affected the scan chain integrity. This was solved by also extending the Chain option to add an inverter to the TCK supplied to the negative-edge triggered flip-flops.

In this article, we introduced Fault the first and only practical open-source DFT toolchain compatible with HDL designs. Fault toolchain provides all needed utilities to generate TVs, simulate faults, and insert scan chains. The fault is aiming at filing some of the gaps in the emerging open-source EDA ecosystem. Also, Fault provides all the needed infrastructure for research activities in digital application specific integrated circuits (ASIC) testing.

Future work includes ATPG acceleration by generating TVs algorithmically to improve the coverage as well as the run-time. While the pseudo-random number generator (PRNG) demonstrated the capability of reaching high coverage in a reasonable amount of time, targeted TVs may yield coverage improvement by covering fault sites the RNG failed to detect. Considerations also include adding support for fault collapsing, use of compiled HDL simulators such as Verilator [12]. We have plans for supporting BIST for memory and logic, TVs compression, and scanchain reordering. Also, we are planning to extend Fault to support a larger variety of fault models like the transition fault model and to add support for fault diagnostics to locate defects and improve the yield. Additionally, we plan to extend fault to support testability checking to detect DFT violations like the generated clock and combinational feedback loops. Finally, we will be adding support for IEEE P1687 and IEEE 1500 standards. ■

Acknowledgments

The Fault is publicly available under the Apache 2.0 license at https://github.com/Cloud-V/Fault, including the benchmarks used for testing. Full installation and usage instructions are available in the GitHub Wiki and the Readme files. It has been tested to work with macOS 10.15 "Catalina" and Ubuntu 18.04 "Bionic Beaver." Because of the complicated set of dependencies required to run the toolchain, a Docker container based on the latter platform has been made available at https://hub.docker.com/r/cloudv/fault. Fault is part of the CloudV Project at the American University in Cairo (AUC), an initiative to reshape digital design and system-on-a-chip design education around open-source software and cloud technologies.

References

[1] H. K. Lee and S. D. Ha, "On the generation of test patterns for combinational circuits," Dept. Elect. Eng., Virginia Polytech. Inst., Blacksburg, VA, USA, Tech. Rep. 12 93, 1993.

[2] *IEEE Standard for Test Access Port and Boundary-Scan Architecture*, IEEE Standard 1149.1-2013 (Revision of IEEE Std 1149.1-2001), 2013.

[3] C. Wolf and J. Glaser, "Yosys—A free Verilog synthesis suite," in *Proc. Austrochip*, 2013, pp. 1–6.

[4] S. Williams. *Icarus Verilog*. Accessed: Feb. 5, 2020. [Online]. Available: http://iverilog.icarus.com

[5] S. Takamaeda-Yamazaki, "Pyverilog: A Python-based hardware design processing toolkit for Verilog HDL," in *Proc. Int. Symp. Appl. Reconfig. Comput.*, Apr. 2015, pp. 451–460.

[6] A. Krstić and K.-T. Cheng, "Delay fault models," in *Delay Fault Testing for VLSI Circuits*. New York, NY, USA: Springer, 1998, pp. 23–31.

[7] Z. Navabi, "Test pattern generation methods and algorithms," in *Digital System Test and Testable Design Using HDL Models and Architectures*. New York, NY, USA: Springer, 2011.

[8] Chris Lattner and Apple Inc. *The Swift Programming Language*. Accessed: Feb. 5, 2020. [Online]. Available: https://swift.org/

[9] Google Inc. *Swift for Tensorflow*. Accessed: Feb. 5, 2020. [Online]. Available: https://www.tensorflow.org/swift

[10] M. K. Prasad, P. Chong, and K. Keutzer, "Why is ATPG easy?" in *Proc. 36th Annu. ACM/IEEE Design Autom. Conf.*, Jun. 1999, pp. 22–28.

[11] M. Shalan and T. Edwards, "Building OpenLANE: A 130 nm OpenROAD-based tapeout-proven flow," in *Proc. Int. Conf. Comput. Aided Design (ICCAD)*, 2020, pp. 1–6.

[12] Wilson Snyder. *Verilator*. Accessed: Jun. 15, 2020. [Online]. Available: https://www.veripool.org/wiki/verilator.

Manar Abdelatty is a Research Associate with the Computer Science and Engineering Department, The American University in Cairo, Cairo, Egypt. Her research interest includes embedded systems and development of electronic design automation software. Abdelatty has a bachelor's degree from Computer Engineering Department, The American University in Cairo (2020).

Mohamed Gaber is a Research Associate with the Computer Science and Engineering Department, The American University in Cairo, Cairo, Egypt. His research interest includes the development of electronic design automation software. Gaber has a bachelor's degree from Computer Engineering Department, The American University in Cairo (2020).

Mohamed Shalan is an Associate Professor (with tenure) with the Department of Computer Science and Engineering, The American University in Cairo (AUC), Cairo, Egypt. His research interests include OpenSource EDA, embedded systems, Internet of Things (IoT), and low-power computing systems. Shalan has a PhD in computer engineering from Georgia Institute of Technology, Atlanta, GA (2003).

■ Direct questions and comments about this article to Manar Abdelatty, School of Sciences and Engineering, The American University in Cairo, New Cairo 11835, Egypt; manarabdelatty@aucegypt.edu.