# Automatic Enhancement of Cell Models to Enable Conditional Stuck-at Fault Simulation and Pattern Generation

# Automatic Enhancement of Cell Models to Enable Conditional Stuck-At Fault Simulation and Pattern Generation

Reza Khoshzaban*†, Nikolaos I. Deligiannis*, Iacopo Guglielminetti†, Michelangelo Grosso†, Riccardo Cantoro*
* Politecnico di Torino, DAUIN — Turin, Italy † STMicroelectronics — Turin, Italy

***Abstract*—Automatic test pattern generation (ATPG) and fault simulation methods for stuck-at faults (SAFs) cannot accurately capture the behavior of conditional stuck-at faults (CSAFs), which manifest only under specific input combinations. To manage such conditional faults using conventional stuck-at fault simulators and test pattern generators, we propose a methodology that embeds conditional fault behavior directly into the logic description of standard cells. The method operates on conditional stuck-at fault models, which define the output variations of a cell in the presence of specific conditional faults for all input stimuli. Each cell model is translated into a Boolean equation that represent the internal logic conditions required to activate the associated faults. This equation is then integrated into the functional description of the standard cell, without modifying its primary inputs, outputs, or intended behavior. As a result, the augmented standard cell library can reproduce the effects of conditional stuck-at faults while remaining compatible with existing fault simulation and ATPG tools. The proposed method incurs negligible application time, and experiments on several benchmark circuits and cores demonstrate reasonable overhead in test pattern count and CPU time.**

***Index Terms*—Stuck-At Fault, Cell Test Model, Conditional Fault, Test Pattern Generation, Fault Simulation**

## I. Introduction

Ensuring the reliability of modern integrated circuits (ICs) is a fundamental step in semiconductor manufacturing. For decades, the cornerstone of this process has been the classical stuck-at fault (SAF) model, which has served as the industry-standard abstraction for defects due to its simplicity, effectiveness, and the efficiency of the Automatic Test Pattern Generation (ATPG) and fault simulation algorithms built around it.

However, the scaling of semiconductor technologies into nanoscale FinFET eras and beyond has revealed the limitations of this model. The classical SAF model is increasingly insufficient as it fails to represent the complex physical defects—such as resistive shorts, opens, and transistor-level imperfections—that dominate in modern, complex cell structures [1]. This discrepancy between physical defects and the SAF model leads to a gap in test quality, where a design can pass all SAF tests yet still contain defect-driven functional failures.

This challenge has been extensively documented in academia and industry, leading to the development of more advanced defect-aware strategies. Test methods such as N-detect [2] and Gate Exhaustive [3] testing have been proposed to implement such strategies. The most prominent of these is Cell-Aware Testing (CAT) [4], [5]. The CAT methodology moves beyond pin-level faults and meticulously characterizes the impact of realistic physical defects inside each standard cell in the library. A significant finding is that many of these internal defects do not behave as simple SAFs. Instead, they manifest as Conditional Stuck-At Faults (CSAFs)—faults that are only activated and observable when a specific set of input values is applied to the cell [6]. While the CSAF model itself was proposed decades ago to model faults in specific logic structures like PLAs [7], its importance has surged in the modern era as the most accurate way to represent the complex, condition-dependent behaviors identified by CAT flows.

Despite their diagnostic accuracy, CSAFs present a major implementation challenge for the industry. The entire EDA infrastructure for digital testing is built and highly optimized for the simple, input-agnostic SAF model. Consequently, standard ATPG and fault simulation tools cannot process CSAFs. To bridge this gap, current approaches are forced to rely on sub-optimal solutions such as utilizing specialized fault simulators that are often less optimized, or attempting partial mappings of CSAFs to traditional SAFs [8].

This paper proposes a novel methodology enabling traditional, high-performance SAF-based ATPG and fault simulation tools to transparently target CSAFs. Rather than modifying the simulator, we modify the library itself. Our method systematically translates CSAF detection conditions into equivalent Boolean logic, embedding this as internal, test-only circuitry within standard cell models. This augmentation is fully compatible with the original cell interface and creates new internal SAF targets functionally identical to the original CSAFs. The enhanced library allows standard EDA tools to generate efficient CSAF patterns, achieving results identical to specialized flows (e.g.,Cell-Aware ATPG flow).

The remainder of the paper is organized as follows. Section II reviews the background on CSAFs and their role in cell-aware testing. In Section III, we present a detailed description of the proposed methodology. Section IV demonstrates the effectiveness of the approach through application results on several cases. Finally, Section V concludes the paper.

979-8-3195-4235-9/26/$31.00 

## II. BACKGROUND

This section provides essential background on CSAFs and the role of cell-aware testing in advancing fault modeling and test generation methodologies.

### A. Conditional Stuck-At Faults

CSAFs represent a crucial extension of the classical SAF model, necessary to capture the complex fault mechanisms inherent in modern scaled ICs. Unlike traditional SAFs, which represent a simple, stuck-at 0 (SA0) or stuck-at 1 (SA1) condition on a net, CSAFs manifest their defective behavior only when a specific logic condition is met at the inputs of the logic cell. This conditionality reflects a more accurate physical reality where certain defects, such as resistive opens or shorts within the transistor structure or interconnect, only disrupt the cell's function under particular voltage or current states, which correspond directly to specific input patterns.

The characterization of CSAFs typically relies on *detection conditions* that define the minimum required input pattern(s) necessary to excite the fault. Table I represents the detection conditions for a set of CSAFs for a 2-input AND gate with inputs $A$ and $B$ and output $Z$. For example, $\text{CSAF}_1$ is detectable at the output only when input vectors $\langle A = 0, B = 0\rangle$ or $\langle A = 1, B = 1\rangle$ are applied to the cell, respectively. This means that when the fault is present and one of these specific input conditions is met, the cell produces an erroneous output. For instance, in the case of the $\langle A = 0, B = 0\rangle$ condition, the correct output of the AND gate is $Z = 0$; however, as shown in Table III, the activated $\text{CSAF}_1$ causes the cell to output an erroneous $Z = 1$. Conventional fault simulators, designed for the SAF model, are not inherently equipped to handle this conditional activation. As semiconductor technology scales down, defects become increasingly complex and internal to the cell structure, meaning that the proportion of defects behaving as CSAFs continues to rise. Therefore, specialized methodologies, such as the one proposed in this paper, are essential to bridge the gap between complex physical defects and established, high-performance commercial test tools.

**TABLE I:** Detection conditions for CSAFs excerpt for a 2-input AND gate.

| $A$ | $B$ | $Z$ | $\text{CSAF}_1$ | $\text{CSAF}_2$ | $\text{CSAF}_3$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | ✓ | ✓ | ✓ |
| 0 | 1 | 0 | - | - | ✓ |
| 1 | 0 | 0 | - | - | ✓ |
| 1 | 1 | 1 | ✓ | - | - |

### B. Cell-Aware Testing and Detection Matrices

CAT addresses the limitations of classical fault models by leveraging detailed knowledge of the internal structure and electrical behavior of standard cells. This methodology, depicted in Fig. 1, targets faults that occur within the cell itself—such as parasitic resistors, coupling capacitors, shorts, and opens—that traditional boundary-focused fault models may fail to detect.

The characterization process begins with layout extraction, which analyzes the physical layout of each cell to identify all transistors, internal nodes, and, most importantly, critical locations for realistic defects. This includes potential bridges between adjacent routing lines or opens at vias and contacts. This physical data is used to guide the fault injection step, which is performed at the analog (SPICE) level. Rather than modeling a defect as a simple digital '0' or '1', a realistic, resistive short (e.g., 100 Ω) or open (e.g., 10 MΩ) is inserted into the cell's transistor-level netlist.

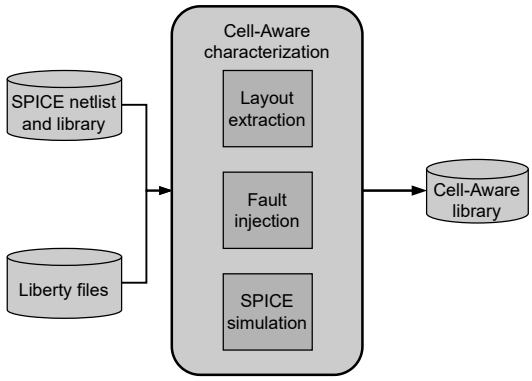


**Fig. 1:** Cell-Aware characterization flow.

When the output waveforms of the defective cell deviate beyond a predefined threshold from the fault-free case, the fault is considered activated for that input condition. These results are systematically recorded in *detection matrices*, which encode, for each input combination, the outputs affected by each defect. An example of the detection matrices for a full-adder cell is reported in Table II. The matrices distinguish between static faults—detectable with 1-timeframe vectors—and dynamic faults, such as slow-to-rise or slow-to-fall transitions that require 2-timeframe vector sequences for detection.

Tables IIa and IIb show the static and dynamic detection matrices, respectively. Each table is organized into three groups of columns: the input values ($A$, $B$, $CI$), the expected output values in the defect-free cell ($CO$, $Z$), and columns representing different defects ($D_2$, $D_3$, etc., for static defects; $D_1$, $D_5$, etc., for dynamic defects).

In the static detection matrix, the numeric values in the defect columns indicate whether and how the defect affects the outputs. Specifically, a value of 1 means the defect manifests on the first output, 2 on the second output, and 3 on both outputs. The rows correspond to different input combinations applied to the full-adder cell and the resulting defect effects.

The dynamic detection matrix follows a similar structure but also includes special symbols such as $R$ (slow-to-rise) and $F$ (slow-to-fall) to represent timing-related defect behaviors caused by consecutive input vectors. These symbols indicate transient faults affecting the outputs differently than static defects.

**TABLE II:** CAT Static and Dynamic Detection Matrices for full-adder cell.

**(a)** Static Detection Matrix

| $A$ | $B$ | $CI$ | $CO$ | $Z$ | $D_2$ | $D_3$ | $D_4$ | $D_7$ | $D_{12}$ | ... |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 2 | 0 | 2 | 1 | 0 | ... |
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | ... |
| 0 | 1 | 0 | 0 | 1 | 2 | 0 | 3 | 0 | 0 | ... |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 3 | 0 | 0 | ... |
| 1 | 0 | 1 | 1 | 0 | 2 | 0 | 3 | 0 | 0 | ... |
| 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | ... |
| 1 | 1 | 1 | 1 | 1 | 2 | 0 | 2 | 1 | 0 | ... |

**(b)** Dynamic Detection Matrix

| $A$ | $B$ | $CI$ | $CO$ | $Z$ | $D_1$ | $D_5$ | $D_6$ | $D_8$ | $D_{42}$ | ... |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | R | 0 | R | 0 | 0 | 1 | 1 | 1 | ... |
| 0 | R | 0 | 0 | R | 0 | 1 | 0 | 0 | 0 | ... |
| R | 0 | 0 | 0 | R | 0 | 1 | 0 | 2 | 1 | ... |
| 0 | 0 | F | 0 | F | 0 | 0 | 0 | 0 | 0 | ... |
| 1 | 0 | R | R | F | 3 | 0 | 0 | 0 | 0 | ... |
| 1 | 1 | 0 | R | F | 0 | 0 | 0 | 0 | 0 | ... |
| 1 | 1 | 1 | 1 | R | 0 | 0 | 0 | 0 | 0 | ... |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |

## III. METHODOLOGY

The proposed methodology, illustrated in Fig. 2, provides a novel, automated flow to convert the complex task of CSAF detection into a standard SAF detection problem. This approach's core advantage lies in its offline augmentation of the standard cell library, which enables the direct use of existing, highly-optimized industrial EDA tools for ATPG and fault simulation. This process avoids the need for specialized simulators or complex changes to the main test flow. It involves two primary stages: 1) translating the abstract CSAF detection conditions into concrete Boolean logic and 2) integrating this new logic directly into the cell's functional description as internal, test-only circuitry.

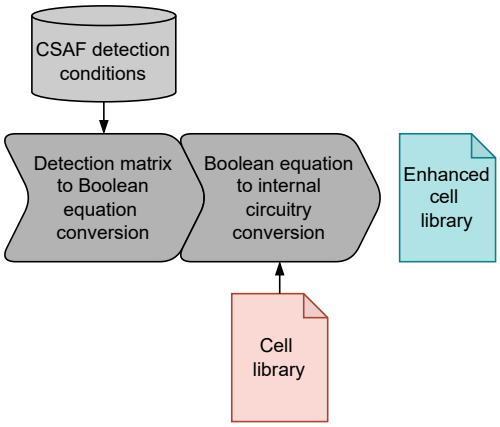


**Fig. 2:** Cell library enhancement flow.

### A. Boolean Equation Derivation

Table III represents truth tables in presence of each CSAF for the example in Table I. For any input vector where a CSAF is not active, the output remains identical to the original fault-free output. However, for any input vector where the CSAF is active, the output value is defined by the characterized faulty behavior (exhibited by the symbol ϟ), effectively modeling the physical defect's impact on the cell's logic. These tables thus define the exact output $Z$ in the presence of the fault for all possible input combinations.

**TABLE III:** Truth tables for each CSAF

**(a)** $Z_{\text{CSAF}_1}$

| $A$ | $B$ | $\text{CSAF}_1$ | $Z_{\text{CSAF}_1}$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 ϟ |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 ϟ |

**(b)** $Z_{\text{CSAF}_2}$

| $A$ | $B$ | $\text{CSAF}_2$ | $Z_{\text{CSAF}_2}$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 ϟ |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

**(c)** $Z_{\text{CSAF}_3}$

| $A$ | $B$ | $\text{CSAF}_3$ | $Z_{\text{CSAF}_3}$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 ϟ |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 ϟ |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 ϟ |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

Each truth table (e.g., Tables IIIa to IIIc) is then systematically translated into its corresponding Boolean equation using standard synthesis techniques (e.g., Sum-of-Products derivation). This translation yields the individual fault-behavior equations for $Z_{\text{CSAF}_1}$, $Z_{\text{CSAF}_2}$, and $Z_{\text{CSAF}_3}$ as follows:

$$Z_{\text{CSAF}_1} = \overline{A} \cdot \overline{B} \cdot \text{CSAF}_1 + A \cdot B \cdot \overline{\text{CSAF}_1}$$
$$Z_{\text{CSAF}_2} = A \cdot B + \overline{A} \cdot \overline{B} \cdot \text{CSAF}_2$$
$$Z_{\text{CSAF}_3} = \text{CSAF}_3 + A \cdot B$$

These individual equations are then combined with the cell's original fault-free logic to create a single, unified cell model that can represent all characterized CSAFs. This synthesis results in a single, comprehensive augmented Boolean equation, $Z_{\text{AUG}}$, as shown in Eq. (1), which is then optimized using a Boolean minimization algorithm (e.g., Espresso) during implementation to ensure the internal logic remains compact. This equation is a multiplexed composite of the fault-free behavior (which is active when all CSAF signals are 0) and the various fault-activated conditions.

$$Z_{\text{AUG}} = \text{CSAF}_3 + \overline{A} \cdot \overline{B} \cdot \text{CSAF}_2 + \overline{A} \cdot \overline{B} \cdot \text{CSAF}_1 + A \cdot B \cdot \overline{\text{CSAF}_1} \quad (1)$$

### B. Standard Cell Augmentation

This composite equation, $Z_{\text{AUG}}$, is then synthesized into a standard gate-level circuit. Fig. 3 provides a schematic of this internal logic, which we refer to as the Internal CSAF

Circuitry. This logic is not an abstraction; it is the actual combinatorial logic that will be added to the cell's definition.

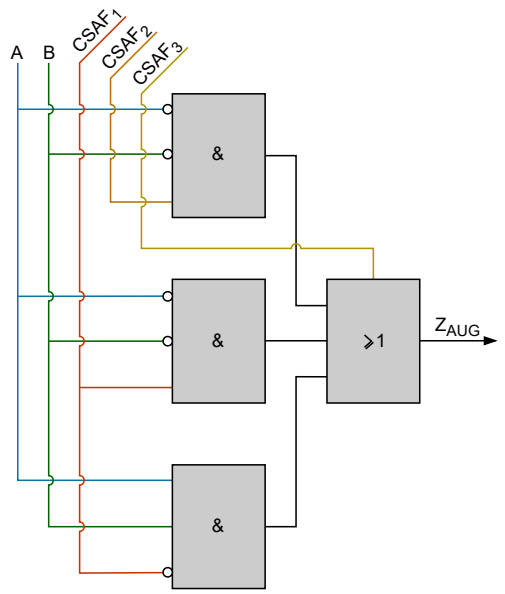


**Fig. 3:** Augmented AND cell's logic diagram with IEC gate notation.

This Internal CSAF Circuitry is then embedded within the original standard cell's functional description, as depicted in the block diagram in Fig. 4. The most critical aspect of this integration is that the modification is entirely internal and hermetic. The cell's external interface—its primary inputs (A, B) and output (Z)—remains unchanged. This ensures 100% pin-compatibility with the original cell library, making the enhanced library a "drop-in" replacement.

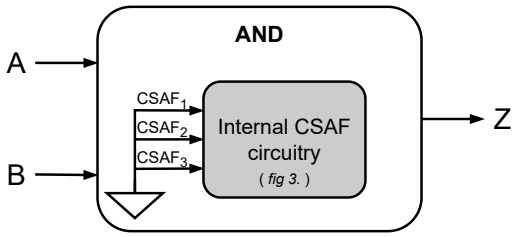


**Fig. 4:** Modified cell overview.

The CSAF nets ($CSAF_1$, $CSAF_2$, $CSAF_3$) are treated as new internal nets that act as control signals for the embedded fault logic. In the fault-free operational mode, these nets are tied to a static logic 0, as shown in Fig. 4. This choice ensures that $Z_{AUG}$ simplifies to the cell's original, fault-free logic, making the circuitry dormant during normal operation.

The CSAF is then activated for testing by targeting the corresponding internal net with a standard SAF. For example, to test for $CSAF_1$, the ATPG tool is configured to target a SA1 fault on the internal $CSAF_1$ net. The tool now has a standard SAF problem to solve: it must find an input pattern that 1) excites the SA1 fault on the $CSAF_1$ net and 2) propagates the fault effect to an observable output (Z). To achieve objective (1), the tool must sensitize the path to the $CSAF_1$ net. By the very design of the Internal CSAF Circuitry, this sensitization path is the CSAF condition itself (in this case, $\langle A = 1, B = 1 \rangle$ or $\langle A = 0, B = 0 \rangle$ (from Table I). The tool, therefore, automatically generates the exact input vector required to test the CSAF, allowing it to be modeled, simulated, and targeted using any standard SAF-based ATPG or fault simulation tool.

## IV. Experimental Results

To assess the effectiveness and scalability of the proposed methodology, we performed extensive experiments on a diverse set of digital designs. The benchmark suite includes combinatorial circuits from ISCAS'85, sequential circuits from ITC'99, ISCAS'89, and three RISC-V processor cores.

### A. Automation Framework

The proposed methodology is implemented as a fully automated software framework developed in Python. The tool bridges the gap between abstract fault models and hardware implementation by processing standard Cell Test Model (CTM) files, which are typically generated by library characterization tools such as Synopsys `CMGen`. In our analysis, we have selected CSAFs from static detection matrices of the CTMs which are relevant from the defect-oriented point of view.

The automation flow, which converts detection conditions into synthesizable hardware, proceeds in three distinct stages:

*1) Parsing and Extraction:* The script first ingests the CTM data to parse the *static* detection matrices for each cell in the library. It automatically identifies the cell's I/O interface and extracts the specific sequential states required for fault activation. The tool constructs a mapping that links every characterizable defect to the specific input vector or sequence required to excite it.

*2) Logic Minimization:* A naive translation of detection matrices into boolean logic would result in excessive area overhead. To mitigate this, our framework employs the `PyEDA` library to perform heuristic logic minimization. The tool converts the raw activation conditions for each defect into Sum-of-Products (SoP) expressions and then minimizes them using the *Espresso* algorithm. This ensures that the internal CSAF circuitry added to the cell is as compact as possible.

*3) Verilog Generation:* In the final step, the tool automatically generates a synthesizable Verilog wrapper for each cell. This wrapper instantiates two modules: 1) a *Table Module* that contains the minimized fault-activation logic, and 2) a *Selector Module* that wraps the original cell, wiring the new internal fault nets (e.g., $CSAF_1$, $CSAF_2$) to the fault logic. This process effectively "upgrades" the standard library to a CSAF-compliant library without manual intervention.

### B. Experimental Setup

We synthesized all designs using the Silvaco Open-Cell 45nm FreePDK [9]. Logic synthesis was performed using Synopsys `Design Compiler`. For the baseline comparison, we generated standard SAF patterns. For the CSAF experiments, we swapped the standard library for our automatically augmented library and targeted the newly created internal fault injection sites. All ATPG and fault simulation tasks were performed using a commercial fault simulation and ATPG tool on an x2 Intel(R) Xeon(R) Gold 6238R machine with 252GB of RAM.

### C. Benchmark Analysis

Tables IV to VI present the comparative ATPG results for standard SAFs versus the proposed CSAF flow.

A key finding is that the pattern count for CSAFs is often comparable to, or even lower than, that of SAFs (e.g., B12 required 221 SAF patterns vs. 199 CSAF patterns). While CSAFs impose stricter activation conditions locally, the high density of these faults within cells often allows the ATPG to serendipitously cover multiple internal defects with the same vector, leading to efficient compaction. Furthermore, comparing our results against a reference cell-aware simulation flow confirmed that our method achieves identical coverage and pattern counts, validating that our "library augmentation" approach is a mathematically equivalent transformation of the problem.

The `C6288` benchmark (a 16-bit multiplier) highlights the impact of circuit topology. Here, the CSAF count (14k) is nearly triple the SAF count (4.7k), reflecting the complexity of the arithmetic cells. This density makes satisfying the conditional constraints significantly harder, resulting in a coverage drop to 90.2% and increased CPU time. This suggests that while the method is efficient for general logic, highly dense arithmetic blocks may require partitioned testing or higher ATPG effort limits.

### D. RISC-V Core Analysis

To evaluate the methodology on RISC-V designs, we targeted the `RI5CY` [10], the `Wally` [11], and `cv32e40p` [12] RISC-V cores. After logic synthesis, all cores were converted to their SCAN equivalent circuits. The results are summarized in Table VII.

For the `Wally` core, we achieved excellent test coverage (99.7%) for CSAFs, virtually identical to the SAF baseline. The pattern count increased moderately from 3k to 4k, demonstrating that for certain microarchitectures, the conditional constraints can be satisfied with efficient pattern compaction. However, the CPU time increased noticeably (from 2.5s to 408s), again reflecting the higher complexity of the search space required to sensitize internal conditional paths compared to standard pin faults.

For the `RI5CY` core, we observe a significant increase in test pattern count (from 2.5k for SAF to 11.7k for CSAF) and CPU time (from 34s to over 2000s). This overhead is expected and indicative of the methodology's rigorousness. Unlike simple pin-level faults, internal cell defects in a processor's datapath often require specific opcode combinations and operand values to be activated (the "condition"). Finding a path that simultaneously satisfies these local cell conditions and propagates the effect through the pipeline requires significantly deeper ATPG search depths, leading to the increased computational cost.

The `cv32e40p` core presents a different insight. While SAF coverage is near-perfect (99.9%), the CSAF coverage drops to 79.4%. This discrepancy is a critical finding. It suggests that a large number of physical defects located deep within the logic cells of this core are effectively untestable under standard functional constraints—they are "conditionally redundant." The standard SAF model optimistically marks the pins as testable, masking this reality. Despite the lower coverage, the tool was able to process the design using standard flows, proving the method's viability for complex, sequential processor cores.

## V. CONCLUSION

We presented an automated methodology to enable standard SAF tools to transparently target complex CSAFs. By embedding fault activation conditions directly into standard cell models as internal Boolean logic, we convert the CSAF detection problem into a conventional SAF one, ensuring full compatibility with existing industrial flows. Experimental results on benchmark circuits and RISC-V cores demonstrate that our approach achieves high test coverage identical to specialized tools for Cell-Aware ATPG flow while incurring reasonable overhead in pattern count and CPU time. Future work will extend this augmentation strategy to model dynamic faults, such as slow-to-rise and slow-to-fall transitions.


## ACKNOWLEDGMENT

This publication is part of the project PNRR-NGEU which has received funding from the MUR – DR 117/2023.


## REFERENCES


[1] F. Hapke et al., "Defect-Oriented Cell-Aware ATPG and Fault Simulation for Industrial Cell Libraries and Designs," in *International Test Conference*, 2009.

[2] I. Pomeranz et al., "On n-detection test sets and variable n-detection test sets for transition faults," in *VLSI Test Symposium*, 1999.

[3] Y. C. Kyoung et al., "Gate exhaustive testing," in *IEEE International Conference on Test*, 2005.

[4] F. Hapke et al., "Cell-Aware Test," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 9, 2014.

[5] F. Hapke et al., "Defect-Oriented Cell-Internal Testing," in *IEEE International Test Conference*, 2010.

[6] F. Hapke et al., "Cell-Aware Production Test Results from a 32-nm Notebook Processor," in *IEEE International Test Conference*, 2012.

[7] O. E. Cornelia, "Conditional Stuck-At Fault Model for PLA Test Generation," 1987.

[8] R. Khoshzaban et al., "Exploiting the Correlation With Traditional Fault Models to Speed-Up Cell-Aware Fault Simulation," in *2025 IEEE International Test Conference (ITC)*, 2025.

[9] Silvaco, *Open-Cell 45nm FreePDK*, https://si2.org/open-cell-library/.

[10] ETH Zurich and Università di Bologna, *PULPino microcontroller system*, https://github.com/pulp-platform/pulpino, 2022.

[11] OpenHW Group, *CORE-V Wally (CVW)*, https : / / github . com / openhwgroup/cvw, 2023.

[12] M. Gautschi et al., *Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices*, https://github.com/openhwgroup/cv32e40p, 2017.

**TABLE IV:** ATPG Results on ITC'99 Circuits

| Design | Stuck-at Faults | | | | | Conditional Stuck-at Faults | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | # Faults | Test Cov. | # Patterns | Test Cycles | CPU Time | # Faults | Test Cov. | # Patterns | Test Cycles | CPU Time (s) |
| B01 | 224 | 100% | 18 | 134 | 0.00 | 90 | 100% | 16 | 120 | 0.01 |
| B02 | 152 | 100% | 15 | 97 | 0.00 | 37 | 100% | 11 | 73 | 0.00 |
| B03 | 922 | 100% | 37 | 1,217 | 0.00 | 283 | 100% | 33 | 1,089 | 0.01 |
| B04 | 2,516 | 100% | 83 | 5,713 | 0.00 | 910 | 99.5% | 68 | 4,693 | 0.03 |
| B05 | 2,738 | 100% | 133 | 4,825 | 0.00 | 1,173 | 96.4% | 108 | 3,925 | 0.18 |
| B06 | 324 | 100% | 20 | 211 | 0.00 | 110 | 100% | 17 | 181 | 0.01 |
| B07 | 1,860 | 100% | 74 | 3,451 | 0.00 | 1,006 | 100% | 75 | 3,497 | 0.09 |
| B08 | 825 | 100% | 61 | 1,427 | 0.00 | 324 | 100% | 54 | 1,266 | 0.00 |
| B09 | 865 | 100% | 49 | 1,501 | 0.00 | 286 | 100% | 37 | 1,141 | 0.00 |
| B10 | 958 | 100% | 59 | 1,141 | 0.00 | 368 | 100% | 46 | 894 | 0.00 |
| B11 | 1,876 | 100% | 102 | 3,297 | 0.00 | 1,171 | 100% | 77 | 2,497 | 0.27 |
| B12 | 5,666 | 100% | 221 | 26,863 | 0.00 | 2,230 | 100% | 199 | 24,201 | 0.00 |
| B13 | 1,582 | 100% | 58 | 2,774 | 0.01 | 525 | 99.8% | 51 | 2,445 | 0.02 |
| B14 | 17,026 | 100% | 440 | 95,698 | 0.05 | 9,977 | 99.5% | 402 | 87,452 | 7.68 |
| B15 | 31,316 | 99.4% | 651 | 271,885 | 0.35 | 16,929 | 98.9% | 773 | 322,759 | 21.64 |
| B17 | 97,292 | 99.2% | 1,565 | 2,056,159 | 1.02 | 51,933 | 97.5% | 1,916 | 2,517,022 | 70.56 |
| B18 | 45,380 | 100% | 858 | 635,661 | 0.06 | 26,055 | 99% | 973 | 720,761 | 7.10 |
| B19 | 92,540 | 99.9% | 1,555 | 2,229,69 | 0.17 | 54,665 | 99% | 1,612 | 2,384,015 | 35.00 |
| B20 | 36,297 | 100% | 750 | 324,433 | 0.12 | 22,178 | 99.7% | 725 | 313,633 | 18.61 |
| B21 | 36,454 | 100% | 763 | 330,049 | 0.12 | 22,463 | 99.6% | 752 | 325,297 | 19.10 |
| B22 | 54,404 | 100% | 1,049 | 645,751 | 0.19 | 32,312 | 99.8% | 997 | 613,771 | 28.76 |

**TABLE V:** ATPG Results on ISCAS'85 Circuits

| Design | Stuck-at Faults | | | | | Conditional Stuck-at Faults | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | # Faults | Test Cov. | # Patterns | Test Cycles | CPU Time | # Faults | Test Cov. | # Patterns | Test Cycles | CPU Time (s) |
| C17 | 46 | 100% | 6 | 8 | 0.00 | 14 | 100% | 4 | 6 | 0.00 |
| C432 | 744 | 100% | 72 | 114 | 0.00 | 323 | 100% | 56 | 112 | 0.00 |
| C499 | 1,244 | 100% | 77 | 154 | 0.00 | 904 | 100% | 130 | 260 | 0.13 |
| C880 | 1,470 | 100% | 75 | 150 | 0.00 | 663 | 100% | 58 | 116 | 0.04 |
| C1355 | 1,238 | 100% | 79 | 158 | 0.00 | 897 | 100% | 130 | 260 | 0.13 |
| C1908 | 1,332 | 100% | 79 | 158 | 0.00 | 868 | 100% | 88 | 176 | 0.12 |
| C2670 | 2,777 | 100% | 157 | 314 | 0.01 | 1,010 | 100% | 127 | 254 | 0.08 |
| C3540 | 3,782 | 100% | 182 | 364 | 0.00 | 1,867 | 99.9% | 152 | 304 | 0.00 |
| C5315 | 5,482 | 100% | 116 | 232 | 0.00 | 2,658 | 99.9% | 106 | 212 | 0.00 |
| C6288 | 4,772 | 100% | 38 | 76 | 0.00 | 14,121 | 90.2% | 91 | 182 | 39.19 |
| C7552 | 5,594 | 100% | 156 | 312 | 0.00 | 2,865 | 100% | 160 | 320 | 0.00 |

**TABLE VI:** ATPG Results on ISCAS'89 Circuits

| Design | Stuck-at Faults | | | | | Conditional Stuck-at Faults | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | # Faults | Test Cov. | # Patterns | Test Cycles | CPU Time | # Faults | Test Cov. | # Patterns | Test Cycles | CPU Time (s) |
| s27 | 90 | 100% | 10 | 56 | 0.00 | 15 | 100% | 11 | 61 | 0.00 |
| s382 | 756 | 100% | 37 | 875 | 0.00 | 252 | 100% | 31 | 737 | 0.01 |
| s420 | 698 | 100% | 55 | 1,009 | 0.00 | 242 | 100% | 47 | 865 | 0.01 |
| s641 | 780 | 100% | 52 | 902 | 0.00 | 260 | 100% | 41 | 715 | 0.01 |
| s713 | 778 | 100% | 54 | 936 | 0.00 | 260 | 100% | 42 | 732 | 0.01 |
| s1238 | 2,232 | 100% | 205 | 4,121 | 0.01 | 1,018 | 100% | 152 | 3,061 | 0.10 |
| s1423 | 2,816 | 100% | 102 | 7,829 | 0.01 | 1,868 | 100% | 91 | 6,993 | 0.08 |
| s1488 | 2,405 | 100% | 132 | 1,065 | 0.01 | 1,180 | 100% | 119 | 961 | 0.70 |
| s5378 | 6,129 | 100% | 267 | 47,705 | 0.01 | 2,395 | 100% | 214 | 38,271 | 0.10 |
| s9234 | 4,764 | 100% | 181 | 26,027 | 0.01 | 1,486 | 100% | 147 | 21,165 | 0.08 |
| s13207 | 6,661 | 100% | 208 | 56,849 | 0.01 | 1,722 | 100% | 190 | 51,953 | 0.04 |
| s15850 | 15,108 | 100% | 464 | 210,646 | 0.03 | 4,878 | 100% | 368 | 167,158 | 0.48 |
| s35932 | 45,598 | 100% | 51 | 89,961 | 0.01 | 27,558 | 100% | 54 | 95,151 | 0.23 |
| s38417 | 48,036 | 100% | 1,032 | 1,617,679 | 0.10 | 1,5097 | 100% | 919 | 1,440,721 | 1.49 |
| s38584 | 50,866 | 100% | 632 | 791,251 | 0.05 | 19,684 | 100% | 577 | 722,501 | 0.81 |

**TABLE VII:** ATPG Results on RISC-V Designs

| Design | Stuck-at Faults | | | | | Conditional Stuck-at Faults | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | # Faults | Test Cov. | # Patterns | Test Cycles | CPU Time | # Faults | Test Cov. | # Patterns | Test Cycles | CPU Time (s) |
| Wally | 337,252 | 100% | 3,090 | 33,101,520 | 2.52 | 125,916 | 99.7% | 4,145 | 44,399,515 | 408.59 |
| RI5CY | 294,850 | 99.8% | 2,476 | 3,742,748 | 34.02 | 187,124 | 98.2% | 11,702 | 17,683,234 | 2,240.83 |
| cv32e40p | 198,152 | 99.9% | 2,279 | 2,578,681 | 6.15 | 113,405 | 79.4% | 2,482 | 2,808,274 | 2,008.87 |