Enhancing SDC Verification in SoCs: Heatmap Visualization and Machine Learning Approaches for Optimal Coverage Closure

Seungkyu Baek, Jaein Hong, Moonki Jun, Sungcheol Park
Samsung Electronics Inc., 1-1 Samsungjeonja-ro, Hwaseong-si, Gyeonggi-do, 18488, Korea

***Abstract*-This paper addresses challenges in SDC verification for System-on-Chip (SoC) designs by proposing a novel approach that integrates heatmap visualization and machine learning. We introduce a method to identify verification gaps through heatmaps and optimize regression test suites using machine learning, significantly reducing redundancy and enhancing efficiency. By applying this approach, we achieved a 63% reduction in regression testing time and improved SDC coverage to 100% for timing exception paths in the regression suite. This method advances verification completeness and efficiency, presenting a robust solution for SoC design and verification processes within constrained development timelines.**

I. Introduction

Synopsys Design Constraints (SDC) define critical design constraints such as operating conditions, timing constraints, and timing exceptions, which serve as essential references to ensure the Power, Performance, and Area (PPA) targets of the design. Mismatches between the SDC and the actual design intent can result in potential performance degradation or device malfunction. This mandates the need for a thorough SDC verification.

Traditional SDC verification relies on Gate-Level simulation (GLS), which typically starts later in the development timeline and is highly time-consuming. Consequently, this often leads to significant timing issues being discovered only in silicon, sometimes necessitating costly re-spins. Therefore, performing SDC verification thoroughly and efficiently is crucial for successful chip development.

This paper presents a novel approach to addressing SDC verification challenges by employing dynamic SDC-aware verification in RTL, thereby enhancing efficiency. It also proposes the use of custom heatmap visualization to enhance SDC verification effectiveness and employs Machine Learning (ML) to optimize test suites for coverage closure, offering a sophisticated methodology for System-on-Chip (SoC) verification. Section II describes existing approaches to SDC verification, including GLS and RTL-based SDC-aware simulation methodologies. Section III describes the proposed approaches to enhance SDC-aware simulation and to shorten the verification time using ML- based coverage closure. Section IV explains the experimental results and achievements of this study, and Section V provides a summary of the conclusions and future work.

II. Existing Approaches

Gate-Level Simulation (GLS) is a widely adopted method that operates on a gate-level netlist generated after the physical implementation phase. It is conducted in an environment where timing information is integrated through the annotation of Standard Delay Format (SDF), the output of static timing analysis, and where designers want to ensure that the synthesized netlist accurately represented the intended RTL behavior [1]. Therefore, performing SDC verification using GLS often results in a delayed start, as it typically commences post-physical design. Additionally, GLS consumes significantly more time compared to RTL simulations due to its lower level of abstraction and intensive computational demands. This not only limits the number of tests that can be executed within the development cycle but also makes it challenging to complete even the limited tests. The extended runtime complicates early identification of timing issues, increasing the risk of costly re-spins. Specific timing issues that have historically led to re-spins include setup and hold violations that were only detected during silicon testing, resulting in significant delays in product launches.

In response to these challenges in GLS, Electronic Design Automation (EDA) tools have introduced dynamic SDC-aware verification as a more efficient approach for early RTL simulation. Fig. 1 illustrates how RTL-level SDC verification can advance the start time compared to GLS, allowing issues to be identified earlier in the design process. In this approach, test vectors developed during functional verification at the RTL level are reused for SDC verification, facilitating comprehensive SDC validation through the Metric-Driven Verification (MDV) methodology based on regression testing [2]. Coverage metrics throughout the MDV cycle are illustrated in Fig. 2. By enabling earlier

detection of potential issues, dynamic SDC-aware RTL simulation improves the verification timeline and reduces the workload associated with late-stage corrections.

SDC-aware verification at the RTL level also includes the capability to verify timing exceptions, such as Multi-Cycle Paths (MCP) and False Paths (FP), which are traditionally challenging to address in GLS. For instance, the Synopsys VCS simulator used in this study enables setup violation checks on MCP and FP specified in the SDC during the design elaboration phase, using pre-defined compile options. When violations are detected, the simulator generates error messages, allowing designers to identify and address timing exceptions earlier in the RTL development process. This capability enhances SDC verification by streamlining the process and reducing potential delays in later design stages.

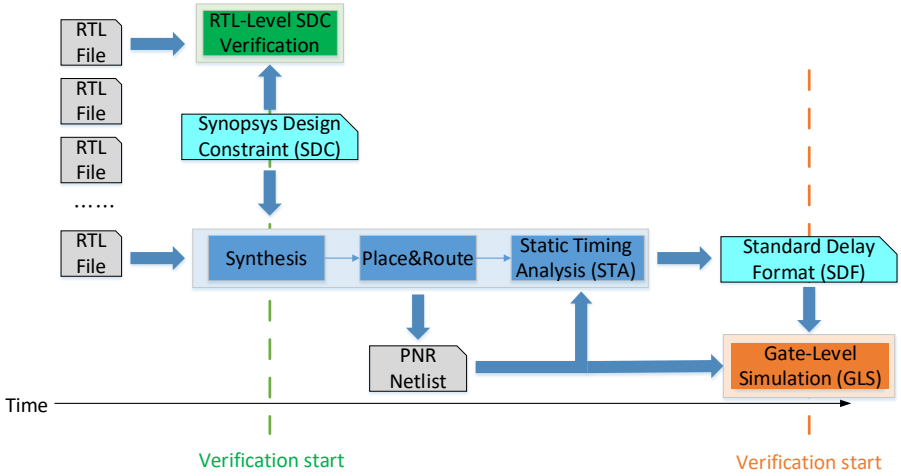

Figure 1. Comparison of verification start time reduction between RTL and GLS.

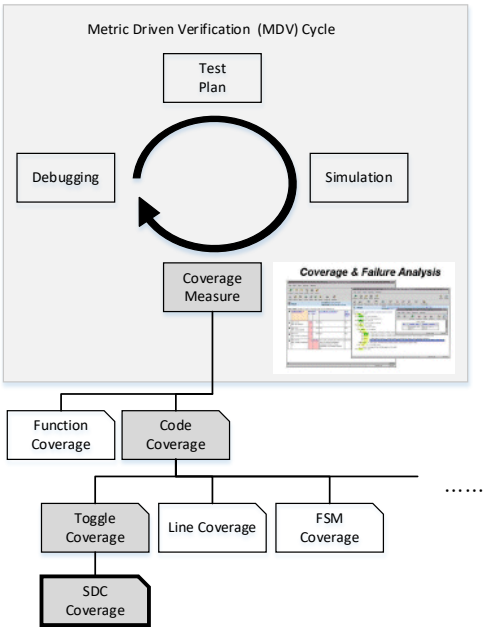

Figure 2. Coverage metrics across Metric Driven Verification (MDV) cycle.

However, despite these advancements, dynamic SDC-aware RTL simulation still faces challenges in achieving comprehensive verification. The inherent X-optimism in RTL simulations, where uninitialized states (X values) can lead to false positives during timing verification, remains a notable issue. To address this, the X-propagation (X-prop) feature can be activated. This feature simulates conditions similar to Gate-Level Simulation (GLS) and allows for early detection of potential timing and data integrity issues. When X-propagation is enabled, the simulation environment adopts GLS-like X-pessimism, propagating injected X values through the design when setup violations

occur. This setup automatically assesses the severity of detected violations through pass/fail outcomes, helping us prioritize critical issues. Failed tests indicate serious timing issues likely to occur in GLS, prompting immediate design adjustments or SDC revisions to prevent costly late-stage re-spins. Conversely, messages from passing tests are often considered false alarms and waived, providing essential information for error investigations through schematic tracing and architecture reviews.

Using this method, we were able to identify key errors, which will be discussed in detail in Section IV, covering experimental results and achievements. However, challenges remain in effectively identifying SDC coverage gaps and optimizing regression testing to reduce redundancy and increase efficiency. To address these issues, this paper proposes two primary solutions: heatmap visualization to target and address SDC coverage gaps, and machine learning-based optimization to improve regression testing efficiency by selecting the most relevant test cases and minimizing redundancy, as detailed in the following section.

III. Proposed Approaches

To resolve the remaining challenges in SDC verification, this paper proposes two key strategies. First, a heatmap visualization method is introduced to effectively identify and address gaps in SDC coverage. By generating heatmaps, we can pinpoint incomplete verification areas, enabling targeted verification actions. Second, we leverage machine learning techniques to optimize regression testing, which enhances testing efficiency by selecting the most relevant test cases and minimizing redundancy. This dual approach accelerates the verification process while ensuring comprehensive coverage. The following sections will elaborate on these methodologies and their impact on verification performance.

A. Efficient Gap Identification in SDC Verification Using Heatmap Visualization

SDC-aware simulation logs and coverage tools typically provide only summarized numerical information, such as source and destination points. This limited information makes it challenging to perform sequential debugging across all required verification paths and ultimately hinders timely verification completion. By implementing heatmap visualization, we can pinpoint areas where verification is incomplete, allowing us to take targeted actions to address these gaps and reduce the risk of undetected timing exceptions and performance degradation in the design.

As shown in the leftmost flowchart in Fig. 3, the process begins with extracting data from simulation logs, which includes information on whether the target paths specified in the SDC have been adequately covered by the tests. Based on this extracted data, we generate an SDC coverage summary table and a heatmap for the target design. The heatmap is structured as a cross-table, where the rows represent source points, and the columns represent destination points, with each cell indicating the test IDs that cover a particular path. A single test may cover one or multiple source-to-destination paths or, in some cases, none at all. This mapping allows us to visually verify the SDC coverage achieved by the regression tests, showing both the number of activated paths and the overall coverage level across the design. On the right side of Fig. 3, heatmap examples illustrate the paths covered by each executed test. The top section of the heatmap provides a summary of the overall test rate, coverage progress, and delta information from previous regression tests. By tracking coverage progression and changes, designers can effectively monitor verification improvements or regressions and identify areas that need additional testing.

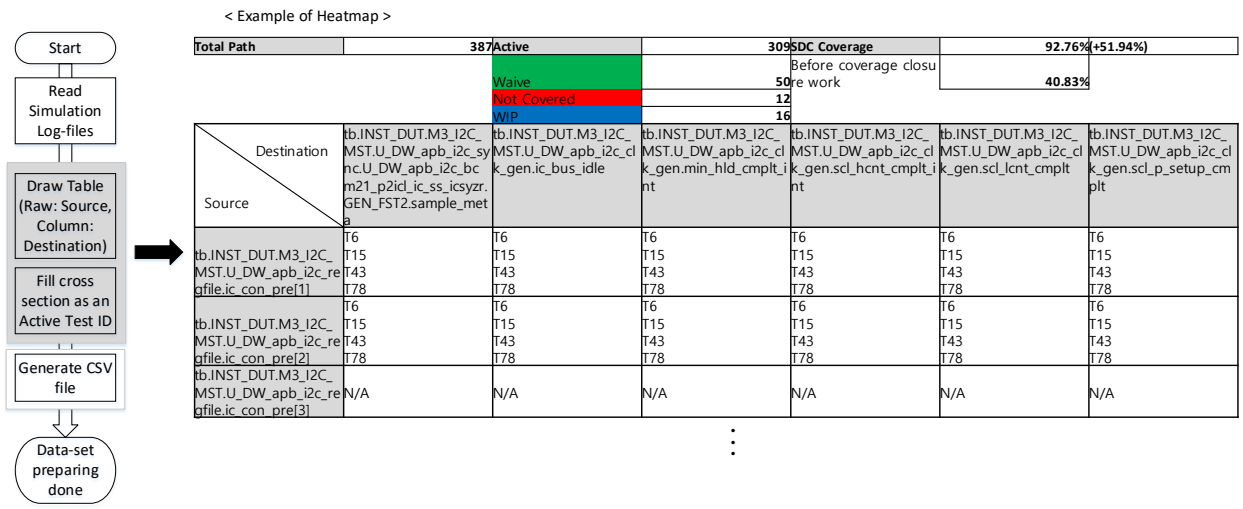

Figure 3. Workflow of heatmap generation with example output.

During the SDC coverage closure process, each cell in the heatmap is assigned a unique color. The initial heatmap is filled with gray and red, including whether a path has been covered or not. Red cells are then changed to green or blue based on the review results, as shown in Fig. 4. Once the coverage closure is complete, only gray and green cells remain in the heatmap.

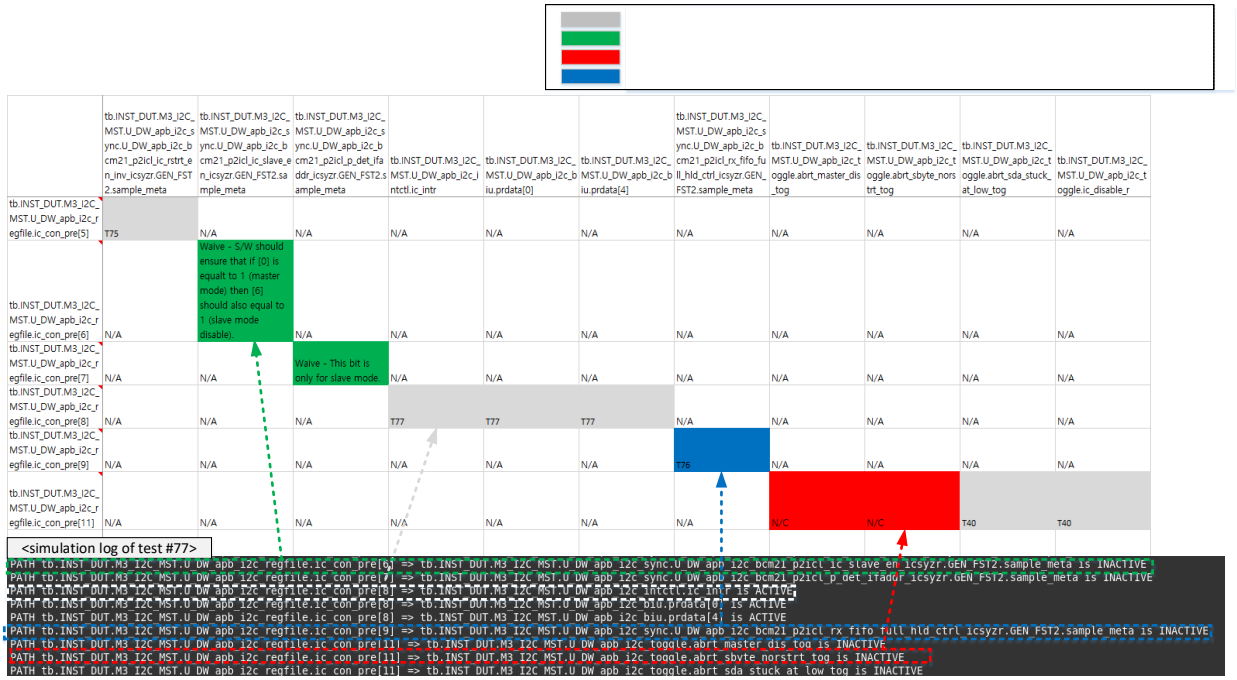

Figure 4. Usage of heatmap for coverage closure.

In addition, data visualization as a heatmap plays an instrumental role in the data preparation stage, which is essential for generating datasets—a prerequisite for machine learning (ML). As shown at the end of the heatmap generation flow on the left side of Fig. 3, the parsed information from raw simulation data is stored in comma-separated value (CSV) files. These CSV files serve as input data for optimization problems and ML models proposed in this article as a methodology to reduce regression and coverage closure turnaround time (TAT). For ML to effectively use this data, raw data undergoes data preprocessing, which includes feature extraction and alignment. For the ML approach suggested to enhance SDC verification, three key features are extracted from the heatmap transformation: (1) the number of asynchronous paths covered by separate tests, (2) the number of unique paths covered by separate tests, and (3) the number of paths that can be waived, such as counter register paths. A sample of the dataset after preprocessing is shown in Fig. 5.

<CSV File #1>
Test ID, Cover Point's Position ID, Simulation Time, Number of Cover Point
T1,DV5,428.26,1
T10,DV5,399.46,1
T14,DV5,429.15,1
T18,DV5,235.84,1
T26,DV5,316.18,1
……
T70,DV5,427.77,1
T72,DV5,760.88,1
T75,DV5,DW6,464.75,2
T76,DV5,EC10,514.63,2
T77,DV5,DZ9,EA9,EB9,391.45,4

<CSV File #2>
Cover Point's Position ID: B2,C2,D2,E2,F2,G2,H2,I2,J2,K2, ……
Is Covered?: 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, ……

Figure 5. Example dataset after feature extraction.

The data generated through heatmap visualization can serve as input for machine learning and optimization algorithms, helping to reduce regression runtime. Since the heatmap is structured based on the paths specified in the SDC, any uncovered fields in the heatmap indicate specific source-to-destination paths that require additional test

scenarios. This highlights the necessity for dedicated test vectors to fully cover SDC constraints and supplement the functional regression suite. To address uncovered paths, we can add directed tests or modify constraints in existing tests to create new vectors that meet the required coverage.

Moreover, heatmap visualization is valuable for waiver decisions when paths specified in the SDC do not align with the design (e.g., due to version mismatches) or represent dead code. By providing a consolidated view of test results based on SDC information, the heatmap assists in identifying paths that may be waived from testing. Specifically, test mappings for each path allow for the extraction of feature datasets that can be used in optimization and machine learning algorithms. These feature datasets include key details such as the coverage scope of individual tests, simulation duration, unique path coverage, and the extent of skippable regions within each test. This information allows us to focus on relevant tests, minimize redundancy, and optimize the regression testing process, accelerating verification while ensuring thorough coverage.

B. Optimizing Regression Testing and Coverage Closure with Machine Learning Techniques

SDC coverage closure involves identifying verification gaps, introducing new scenarios to address these gaps, and iterating through regression tests to verify changes. In the previous section, the heatmap focuses on detecting and closing these gaps. To optimize subsequent regression tests, this paper proposes using machine learning to refine the test suite. Based on coverage results, test priorities are set, and redundant tests are removed. With machine learning, the test suite is optimized, reducing regression time and accelerating coverage closure. The overall coverage closure flow is shown in Fig. 6.

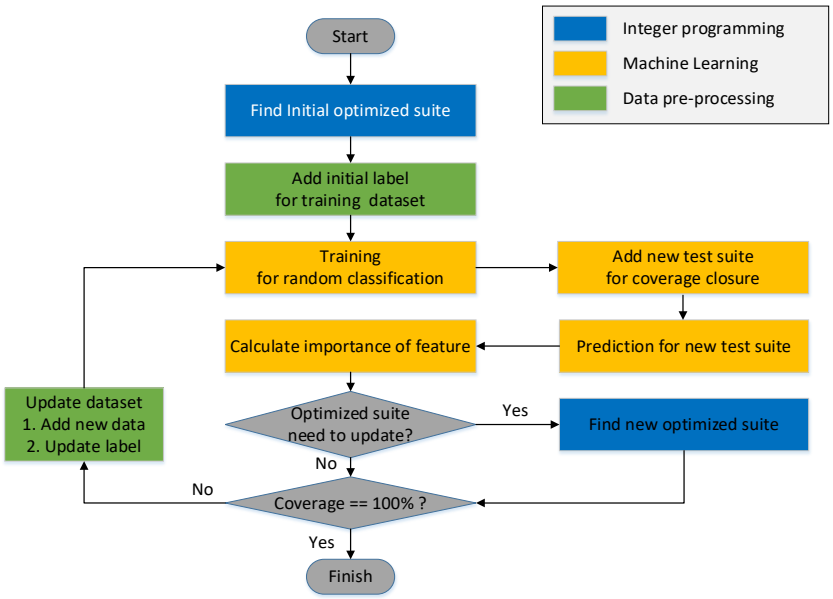

Figure 6. SDC coverage closure flow with optimized test suite finding.

This approach consists of two main components in Python: solving the optimization problem using the PuLP library and training a Random Forest classifier with scikit-learn [3]. The following numbered sections outline each component and the rationale for using PuLP and Random Forest in this context.

1) Role and Rationale for PuLP and Random Forest

The PuLP library is used to formulate and solve linear optimization problems. In this study, PuLP plays a crucial role in optimizing the test suite by selecting a minimal number of tests that can comprehensively cover the required paths while minimizing redundancy. The library's flexibility and ease of use in handling binary and integer variables make it well-suited for modeling test selection as a binary decision (1 if selected, 0 otherwise). By defining an objective function that minimizes the number of selected tests while covering all paths, PuLP effectively prioritizes high-impact tests and excludes redundant ones, thereby optimizing the regression test suite.

The Random Forest classifier predicts the importance of newly added tests and determines whether they should be included in the optimized suite. Random Forest was chosen over other machine learning algorithms due to its robustness, high interpretability, and natural ability to handle feature importance. As an ensemble method that builds multiple decision trees and aggregates their results, Random Forest effectively manages diverse feature sets, such as test execution time, coverage contribution, and skip ratios, without overfitting. With fewer hyperparameters than many other algorithms, Random Forest is also easier to tune for consistent performance in test suite optimization.

2) Objective Function and Constraints in Optimization

The goal of optimization is to minimize the number of selected tests while covering all asynchronous paths. This basic objective function can be written as Formula (1), where x_i represents the binary value of each test: "1" if the test is chosen and "0" otherwise, with n as the total number of tests.

$$\text{Minimize } \sum_i^n x_i \tag{1}$$

In addition, each test's unique paths, skippable paths, and simulation time are weighted, yielding the final objective function in Formula (2), where w_i denotes the weight of each test.

$$\text{Minimize } \sum_i^n w_i x_i \tag{2}$$

This paper considers only explicit constraints, such as binary and path coverage constraints, while excluding implicit constraints. The binary constraint, defined in Formula (3), restricts each test x_i to two possible values: 1 or 0.

$$x_i \in \{0,1\}, \forall i = \{1,2, \dots, n\} \tag{3}$$

The coverage constraint, shown in Formula (4), requires each path to be covered by at least one test, where A represents all paths to cover, and S_i is the subset of paths covered by test i.

$$\sum_{i:j\in S_i} x_i \geq 1\,, \forall j \in A \tag{4}$$

To find an optimized solution that minimizes the sum, the objective function and constraints can be implemented using the PuLP library in Python. Table I provides an example of pseudocode using PuLP. This code minimizes the number of selected tests while ensuring that all required paths are covered, as per Formula (1).

TABLE I
PYTHON CODE EXAMPLE FOR USING PULP OPEN LIBRARY

```
# Create the model
    model = LpProblem(name="minimum-testset-selection", sense=LpMinimize)
# Decision variables: 1 if test is selected, 0 otherwise
    test_vars = {test_id: LpVariable(name=f"test_{test_id}", cat="Binary") for test_id in test_allocations}
# Objective: Minimize the number of selected tests
    model += lpSum(test_vars.values()), "Total_Tests"
# Constraints: Each area must be covered by at least one test
    for a in res_areas:
        model += lpSum(test_vars[t] * (test_weights[t]/adjust) for t in test_allocations if a in test_allocations[t]) >= 1,
f"Path_{a}_constraint"
# Solve the problem
    model.solve()
```

3) Machine Learning Classifier and Feature Extraction

To improve the efficiency of coverage closure, we applied a Random Forest-based machine learning classification approach to reduce the number of regression executions. Random Forest is an ensemble method that trains multiple decision trees randomly to achieve results in both regression and classification tasks. This method is well-suited for our optimization needs, as Random Forest is capable of handling various features with minimal tuning and can provide insights into feature importance, which is valuable in understanding the contribution of each test.

For classifier training, we conducted supervised learning using a dataset including features from prior data preprocessing, as illustrated in TABLE II. The results from selecting the optimized regression suite serve as the ground truth. During functional verification, we predict whether new test scenarios can be added to the optimized regression suite based on the predict_proba values. If the value exceeds 0.5, we re-execute the optimization process and update the regression suite.

TABLE II
SNIPPET OF PYTHON CODE FOR ML TRAINING

```
# Prepare training dataset
X_train = np.column_stack((time_consumption, number_of_covered_points, unique_areas_list, aggregate_skipability_list))
y_train = []
for tid in test_allocations.keys():
    print("{} ... ({})".format(tid,len(test_allocations[tid])))
    if tid in optimized_regression_suite:
        y_train.append(1)
    else:
        y_train.append(0)
# Train the RandomForestClassifier
clf = RandomForestClassifier()
clf.fit(X_train, y_train)
```

By applying this method, we reduce the number of regression executions, thereby decreasing the TAT required for coverage closure. The combination of PuLP for optimization and Random Forest for classification enables a more efficient and streamlined approach to regression test suite management, ensuring timely coverage closure and minimizing redundancy.

IV. Experimental Results and Achivements

This section presents the experimental results and outcomes of applying the proposed SDC-aware verification methodology to a CPU block within a SoC design targeted for High-Performance Computing (HPC) applications. The design includes peripherals such as GPIO, UART, I2C, SPI, QSPI, and a timer, along with ARM Cortex-M3 and ARM Cortex-A55 (Quad-Core) processors. The core operating frequency is 1.5 GHz, and the design comprises approximately 10 million gates fabricated in a 4nm process node.

The SDC for the CPU block consists of 1,069 constraints, including timing exceptions such as multi-cycle paths and false paths, which are the primary focus of this paper. These exceptions are defined using 37 specific commands (e.g., set_multicycle_path, set_false_path) that affect a total of 82,554 paths in the design. In addition to timing exceptions, the SDC also includes other constraint categories such as clock frequency, constant values, I/O delays, and clock groups. To validate the design, 554 test scenarios were implemented for functional regression testing at the CPU block level.

The goal of the verification process was to detect design errors and achieve complete SDC coverage closure. By utilizing an optimized regression suite and applying machine learning techniques, we focused on minimizing the regression test time while ensuring full coverage of critical paths. Key findings from this verification effort include the identification of specific design errors and the steps taken to achieve SDC closure. These results demonstrate the effectiveness of the proposed methodology in improving the verification efficiency and accuracy of SoC designs, ultimately contributing to faster time-to-market and more reliable product outcomes.

A. Identifying Key SDC Errors Early through X-Propagation in RTL Verification

SDC-aware simulation logs and coverage tools typically provide only summarized numerical information, such as source and destination points, which limits detailed debugging across all verification paths. This limited information makes it challenging to identify critical issues at the RTL simulation stage. By activating X-propagation during RTL-level SDC verification, however, we can detect two types of SDC errors that are frequently overlooked before setting up the GLS environment. Both types of errors can lead to repeated mistakes, potentially impacting SoC functionality and development timelines:

1) Timing exception constraints defined at incorrect points
2) Timing exception constraints based on misunderstood clock relationships

Fig. 7 illustrates an example of detecting the first type of error. This error is identified during the "fail" test message debugging with X-propagation activated. It typically occurs when timing exception constraints are incorrectly defined due to misunderstandings of clock relationships. Such errors can significantly impact SoC functionality and require either an SDC or design change, followed by a full regression rerun after the fix. To improve verification efficiency, this error is given priority over others, allowing high-impact issues to be addressed early in the process.

The second type of error is depicted in Fig. 8. This issue is identified during the message cleaning process for "pass" tests. It often arises from misunderstandings in defining timing exceptions for asynchronous paths. While it does not immediately affect SoC functionality, it can be waived if deemed safe during the architectural review. This helps reduce the verification workload and allows engineers to focus on more critical areas, streamlining the overall verification process.

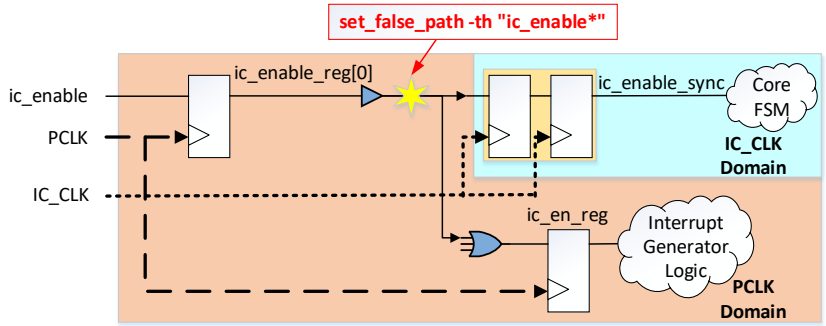

Figure 7. Detecting timing exception errors in "fail" tests with X-propagation.

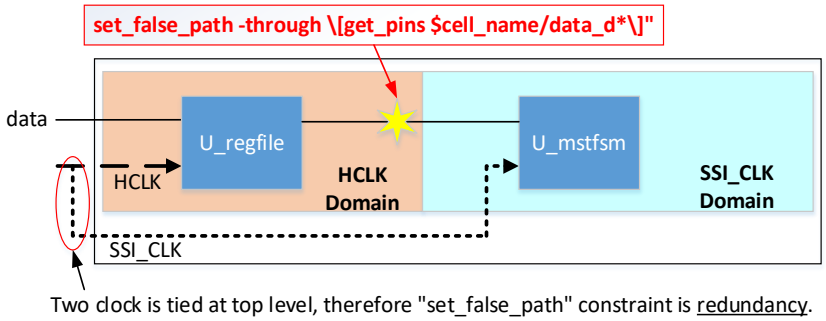

Figure 8. Identifying and waiving asynchronous path errors in "pass" tests

B. Efficiency Gains in SDC Verification through Optimized Test Suites and Machine Learning

The proposed heatmap visualization and machine learning-based optimization were initially applied to timing verification of a specific IP within the SoC, demonstrating measurable efficiency gains. Based on these results, we are currently expanding this approach to the SS block level and examining its potential for broader application across the SoC.

Compared to the target CPU block design described earlier, the timing exception paths extracted for the specific IP were limited to 387. Through SDC coverage closure for specific IP instances, the optimization algorithm effectively reduced the regression test suite from 78 total test vectors to 19, achieving 100% coverage of the timing exception paths defined in the SDC. This optimized suite ensured comprehensive verification of critical timing constraints for the IP instance while streamlining the test set, contributing to overall resource efficiency. The results, including a comparison of the regression suite size and TAT between the proposed approach and the existing method, are summarized in TABLE III.

Using the optimized test suite also reduced regression runtime from 27 hours to 10 hours, achieving approximately a 63% reduction in verification time. This gain in efficiency translates to faster iterative cycles and reduced engineering effort in the verification process, enabling engineers to allocate more time to other critical areas of design validation.

TABLE III
SUMMARY OF REGRESSION TAT REDUCTION ACHIEVEMENTS

	Existing method	Proposed ML based approach
Regression suite size	78 test vectors	19 test vectors
Regression Turn-Around-Time (TAT)	27 hours	10 hours

In comparison, the traditional GLS-based SDC sign-off approach was limited to a single test vector, covering just one path with a minimal coverage score of 0.3%. By applying the proposed method to maximize SDC constraint verification, GLS coverage improved significantly. A single optimized vector verified 203 timing paths, covering approximately 60% of SDC-defined paths. This result demonstrates that the optimized suite enables a more comprehensive validation of timing constraints, particularly at the SS block level, where complex clock relationships and asynchronous paths require robust verification.

V. CONCLUSION AND FUTURE WORK

This study demonstrates that combining heatmap visualization with machine learning-based optimization enhances SDC verification by improving coverage and reducing runtime. The optimized approach streamlines the regression suite, achieving comprehensive timing verification for critical paths while significantly reducing verification time.

The proposed methodology has been evaluated within a limited subset of IPs at the subsystem level. When applied at the block or chip level, several challenges may arise, such as generating heatmaps with thousands of rows and columns and determining the effectiveness of coverage closure based on these heatmaps. Moreover, implementing this methodology at the chip level requires a comprehensive understanding of the system to define complex verification scenarios. In the development process, where SDC verification must be stabilized at the RTL freeze point, replacing the current structural CDC/RDC verification or subjective designer reviews with functional regression for functional verification and metric-based coverage closure provides an opportunity to ensure objective reliability. This approach offers significant advantages in improving verification quality and mitigating risks to the development schedule.

Currently, this methodology is being applied at the block level, specifically for the LPDDR5X controller and PHY, where coverage closure is being performed for an SDC that requires verification of 64,202 paths derived from 318 constraints. In this context, the methodology has demonstrated a reduction in TAT.

Building on these findings, future work will extend this methodology to full-system SoC validation, further leveraging machine learning to create highly optimized and resource-efficient verification suites. Additionally, we will explore AI-based predictive algorithms to dynamically refine the test suite as design complexity grows, ultimately supporting a fully automated SDC verification process. This extension will enhance verification scalability and reliability, contributing to reduced development cycles and faster time to market.

REFERENCES

[1] Gagandeep Singh, Dec 2021, "Gate-Level Simulation Methodology: Improving Gate-Level Simulation Performance", Cadence Design System,Inc. pp. 18-19.

[2] Carter, H. B., & Hemmady, S. G., 2007, Metric Driven Design Verification An Engineer's and Executive's Guide to First Pass Success (1st ed. 2007.). Springer US.

[3] Stuart Mitchell, Michael O'Sullivan, Iain Dunning, "PuLP: A Linear Programming Toolkit for Python" Stuart Mitchell, Michael O'Sullivan, Iain Dunning, "PuLP: A Linear Programming Toolkit for Python".