Streamlining RAL-based Cross-Coverage and Sequence Coverage through Automation

Satyajit Sinari, Vijayakrishnan Rousseau, Ram Immaneni, Mangayarkarasi Arumugam, Prasad Shah, Raghavendra C K
Intel Corporation, Folsom, USA
satyajit.j.sinari@intel.com
vijayakrishnan.rousseau@intel.com
ram.n.immaneni@intel.com
mangayarkarasi.arumugam@intel.com
prasad.s.shah@intel.com
Intel Corporation, Bengaluru, India
raghavendra.c.k@intel.com

***Abstract*- Modern hardware verification increasingly relies on automation to improve efficiency, consistency, and coverage quality. While register-level RTL and UVM Register Abstraction Layer (RAL) models can be automatically generated from register specifications, coverage automation has traditionally focused on field-level bins derived directly from these specs. However, the most critical coverage models are those that stress the Device Under Test (DUT) through concurrency or expose rare corner cases. These coverage points, essential for subsystem and System-on-Chip (SoC) validation, cannot be auto-generated and are often manually coded, leading to redundancy, inconsistency, and inefficiency. This paper presents a novel, modular framework centered around a Graphical User Interface (GUI) that enables verification engineers to efficiently define, manage, and reuse complex coverage models. The proposed framework parses Intellectual Property (IP) register specifications to provide direct access to the register library, and supports intuitive construction of covergroups, coverpoints, cross coverpoints, and sequence-based coverpoints. It accelerates coverage definition by allowing users to save and load models in a machine-readable format, preview generated SystemVerilog code, and integrate coverage models seamlessly into UVM testbenches. The workflow reduces manual errors and enhances productivity. Results show significant improvements in coverage model creation rate, reuse, and integration, thereby supporting robust validation of complex hardware designs.**

I. Introduction

Automation is fundamentally transforming hardware verification, delivering significant gains in both efficiency and consistency. In current verification flows, register-level RTL and Universal Verification Methodology (UVM) Register Abstraction Layer (RAL) models can be automatically generated from register specifications or design documentation, greatly expediting the early stages of verification. This automation also encompasses the creation of coverage points, where valid bins for each register field are derived directly from the specification, ensuring thorough field-level coverage with minimal manual effort.

Despite these advances, defining comprehensive coverage remains a complex task. While automation can handle basic coverage generation, the manual development of covergroups is often necessary to capture intricate scenarios and corner cases, particularly those arising from complex interactions among multiple registers or fields, or from specific sequences of register operations. Such higher-order coverage points become especially critical during advanced validation phases, such as subsystem or System-on-Chip (SoC) integration, where interaction complexity increases.

As hardware designs grow in complexity and the number of registers expands, manual coverage definition becomes increasingly labor-intensive and error-prone. This not only slows down the verification process but also introduces inconsistencies, especially when multiple engineers independently define similar coverpoints and cross-coverage scenarios. The resulting fragmentation complicates the maintenance and reuse of coverage models.

To address these challenges, this paper proposes a Graphical User Interface (GUI)-based framework that abstracts the underlying RAL access syntax and allows verification engineers to interact directly with register specifications to define coverage models, leveraging their familiarity with register and field nomenclature. The backend automatically translates these selections into the appropriate RAL access format, streamlining the generation of coverage models. The GUI supports the definition of covergroups, coverpoints, and cross coverpoints using an intuitive interface that abstracts standard Language Reference Manual (LRM) [1] coverage syntax. As a result, users can define robust and flexible coverage models without deep expertise in UVM or RAL syntax, relying instead on their understanding of the design and register specification.

This GUI-driven workflow accelerates coverage model development, reduces the likelihood of syntactic and semantic errors, and enhances the overall efficiency and reliability of the verification process. Furthermore, the interface enables saving and loading of coverage models in a machine-readable format, promoting reuse and collaboration across verification teams.

II. WORKFLOW

Fig. 1 presents a streamlined workflow for coverage definition in a UVM-based testbench. The CovDefTool Block acts as a central hub, interfacing directly with the user through the GUI, handling backend processing and scripting tasks, and maintaining seamless access to the coverage model database.

1) *GUI Frontend*: The verification engineer uses the GUI to query the register specification and define coverpoints. With the GUI, users can quickly define both coverpoints and cross coverpoints. For sequence coverage models, users can input various sequences of register transactions.

2) *Backend:* The backend parses the register specifications and provides users with direct access to the register library. Once all necessary coverpoints are defined using the GUI, the backend processes them and outputs the coverage model as a SystemVerilog file. This file is seamlessly integrated into the UVM testbench coverage block, completing the workflow.

3) *Coverage Model Management:* The tool also supports saving coverage models in a machine-readable format, which can be loaded, reused, extended, or modified by users.

Overall, this workflow emphasizes a modular and automated approach, with a simplified tool for coverage definition, machine-readable storage, and code generation. This setup streamlines the user's workflow, reduces manual errors, and accelerates the process of building robust coverage models for complex hardware designs.

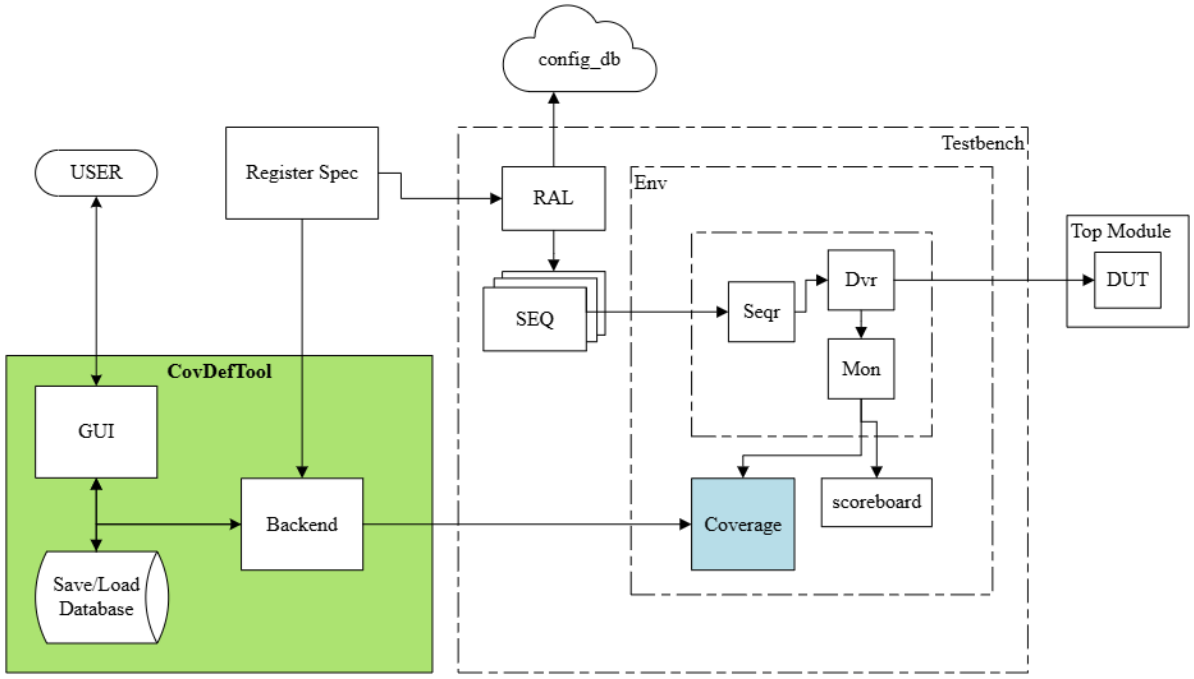

Figure 1. Solution for streamlining definition of Coverage Models

Fig. 2 illustrates the CovDefTool Block workflow, which serves as an integrated module encompassing both the GUI and backend functionalities, in addition to managing the coverage model database. This integrated design streamlines user interaction, automates processing tasks, and maintains robust data handling for coverage model validation workflows.

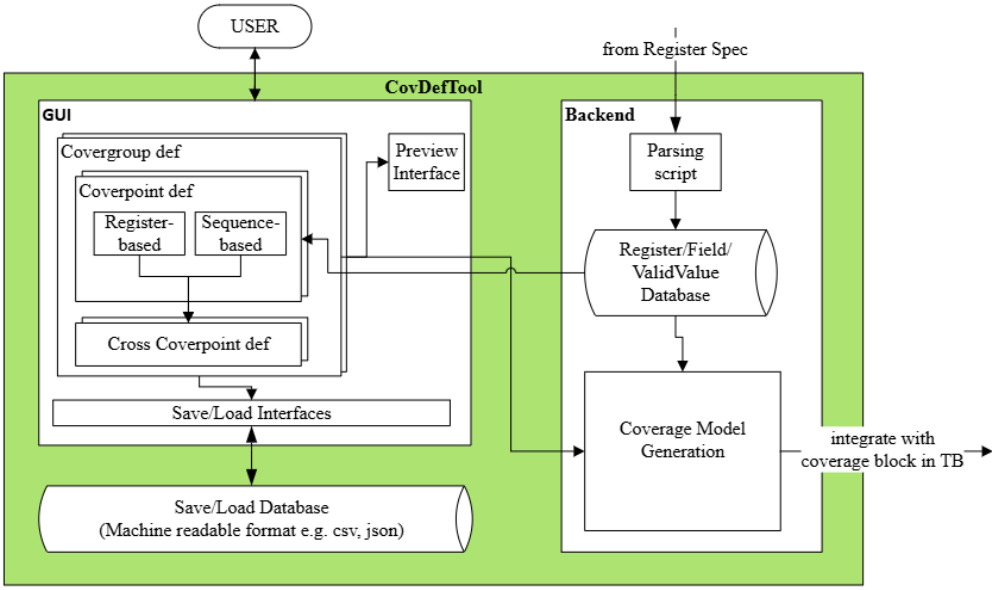

Figure 2. GUI FrontEnd and BackEnd details

A. GUI Frontend Block

Fig. 3 presents the GUI window, or the frontend, for the CovDefTool, which consolidates the various interface components detailed in a subsequent section. This front-end interface enables users to interactively define, save, and load coverage models, thereby facilitating the configuration and management of validation testbenches.

1) *Register Search:* The user can search for any register defined in the IP register specification, and some level of wildcard search is supported. Once the register is selected, all the fields in the register and valid bins (based on the legal values for the field) are populated in the tool.

2) *Register Coverpoint Definition:* The user can now pick the field(s) of interest and designate them as coverpoints. The bins of interest can be specified by retaining those directly derived from spec-defined valid values for a given register field, or the user may add or delete any desired bin. Additionally, ignore and illegal bins can be specified. Constructs such as iff and with are supported as defined in the LRM [1]. This accelerates coverpoint definition by offering rapid lookup of registers and fields directly from the register specification and allowing modifications as required.

3) Cross Coverpoint Definition: After the required register fields are defined as coverpoints with the appropriate bins, the GUI streamlines cross definition by listing previously defined coverpoints (from the Coverpoint Definition interface) along with their associated bins, which can be selected from a dropdown. This reduces manual errors and increases flexibility. Additionally, the tool allows the user to define bins (valid, ignore, and illegal) for the cross coverpoint. It supports the majority of LRM [1] defined cross coverage constructs, including iff, with, intersect, and select expression. This interface enables users to define cross coverage effectively without requiring deep knowledge of the exact syntax.

4) *Sequence Coverpoint Definition:* Often, the order in which register configurations are programmed can result in different IP behaviors or help uncover interesting corner cases. To define such coverpoints, this tool introduces a new ‘sequence coverpoint,’ which lets the user define a sequence of RAL-based register transactions that is compared against input stimulus to collect coverage.

5) *Preview Interface:* An integrated preview window lets users inspect the generated coverage code with translated syntax prior to testbench integration.

6) *Covergroup Definition:* The tool allows users to assign a covergroup name and associate various coverpoints with it. Multiple covergroups can be defined for different coverpoints. All covergroups will be encapsulated into a single class at the time of code generation by the backend. It also supports covergroup parameterization and sampling function parameterization. An option to override the sampling function with event triggers is also provided to the user.

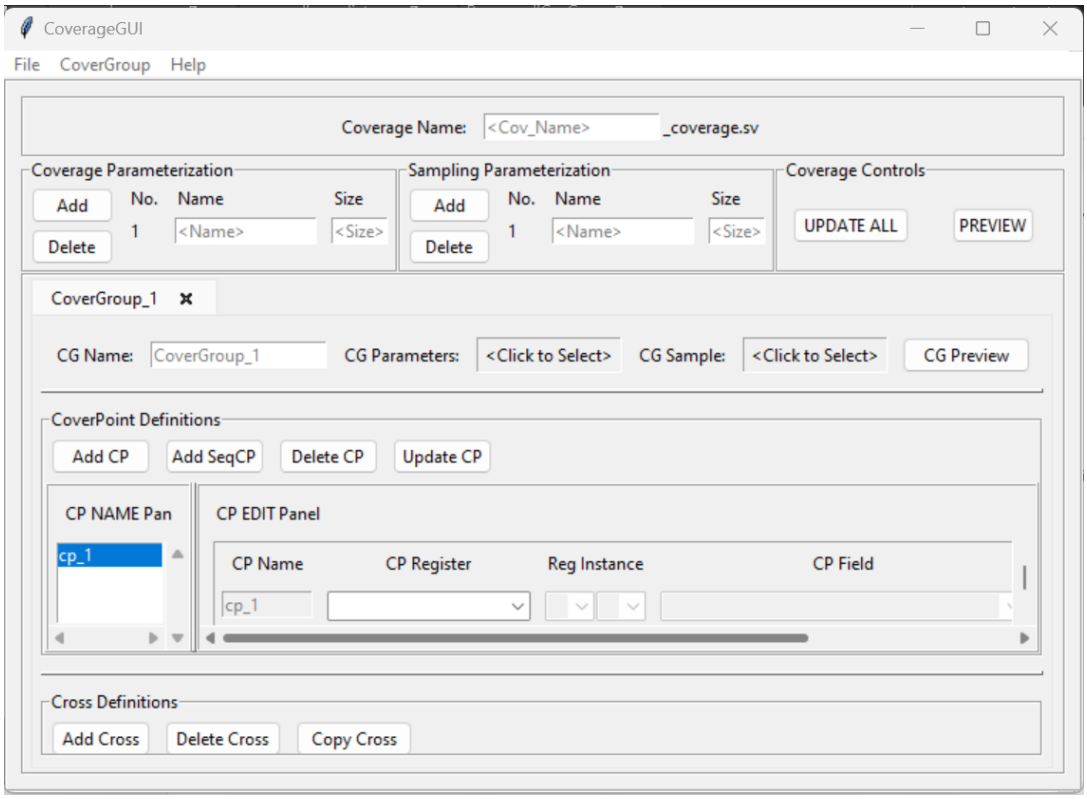

Figure 3. The CovDefTool GUI Window

B. The Backend Block

In Fig. 2, the architecture and workflow of the GUI Backend block are depicted in detail. The backend block is responsible for parsing the register specification, extracting relevant data, and storing this information in an indexed database to facilitate efficient retrieval by the frontend. Subsequently, the Coverage Model Generation block queries the indexed database, combining the extracted register information with user-provided coverage definition details. Leveraging this combined data, the Coverage Model Generation block produces a SystemVerilog file that contains a syntactically accurate coverage class composed of embedded covergroups and coverpoints based on the RAL.

For sequence-based coverpoints, additional logic is required outside the coverage model. It is necessary to capture all input register transactions from the monitor and store them in a queue, as shown in Fig. 4. All user-provided sequence coverpoints are translated into expected queues comprising the sequence of register transactions as requested by the user. At the sampling point, the expected queues are compared with the actual stimulus queue to check if the expected sequence of register transactions took place in the current test. Coverage is captured based on the status bit returned from this compare function.

The backend is designed to be aware of the testbench structure, enabling it to automatically edit the necessary testbench files and seamlessly integrate the coverage model with correct instantiation based on coverage parameters. Sampling functions are overridden with event trigger information provided through frontend and are defined and called at necessary points within the testbench.

This interface allows saving the defined coverage model in a machine-readable formatted file to an external database. Owing to the framework's modularity, users can load existing models to extend or modify coverage. This capability accelerates definition for new features, especially when crossing legacy and newly added coverage, and reduces redundancy.

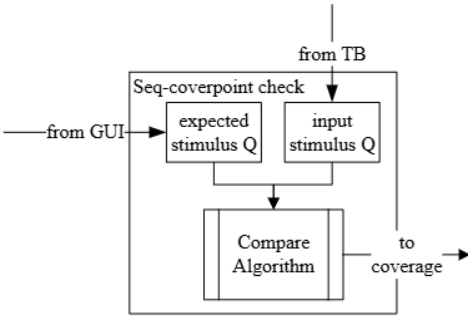

Figure 4. Sample Sequence covergroup and function definition code

III. CASE STUDY

A. RAL-based Coverpoint and Cross Coverpoint Case Study

As a case study, we consider a hypothetical intellectual property (IP) core implementing a Universal Asynchronous Receiver/Transmitter (UART), which is a widely used hardware communication protocol facilitating serial data exchange between two devices. For this IP, a register set is specified, detailing the address offsets and corresponding fields for each UART register. Table I provides an example TX CTRL (Transmission Control) Register Spec with its corresponding access through the pre-defined RAL.

TABLE I
REGISTER SPECIFICATION FOR A SAMPLE TX CTRL REGISTER

Register Name	Address Offset	Fields (bit positions)	Description	RAL access of fields
TX CTRL	0x00, 0x04, 0x08, 0x0C	TX Enable (0) Parity Enable (1)	Controls IP operation and reset.	ral_reg.tx_ctrl_reg[inst].tx_enable ral_reg.tx_ctrl_reg[inst].parity_enable

The RAL model facilitates the integration of functional coverage by enabling the definition of coverpoints and associated bins for each register field. Fig. 5 shows the CoverPoint (CP) definition interface and defines two coverpoints, cp_tx_enable and cp_baud_value, as seen in the CP Name Panel. As each coverpoint is selected, the configuration details for that coverpoint can be viewed and edited in the CP Edit Panel. Valid field values are algorithmically converted into bins: fields with width < 4 are mapped to explicit bins named according to the register specification, while fields with width ≥ 4 are partitioned into three evenly distributed ranges.

Fig. 6 presents the cross coverage interface, which demonstrates a fundamental use case in coverage-driven verification. The interface enables users to select any number of coverpoints from the CP Name Panel window and establish cross coverage among them using the Add Cross button. In the example shown, cross coverage is configured between two previously defined coverpoints. The example is specifically intended to verify that high baudrate values are observed when the transmitter is enabled, thereby ensuring correct functional interaction between these two parameters.

The methodology demonstrated in this example is inherently scalable. It can be extended to encompass additional registers within the same intellectual property (IP) block, as well as across multiple IPs. The GUI facilitates this scalability by enabling access to multiple RALs and their associated registers, thereby supporting comprehensive cross coverage analysis across a broader verification space. Fig. 7 illustrates the final covergroup code generated using the CovDefTool.

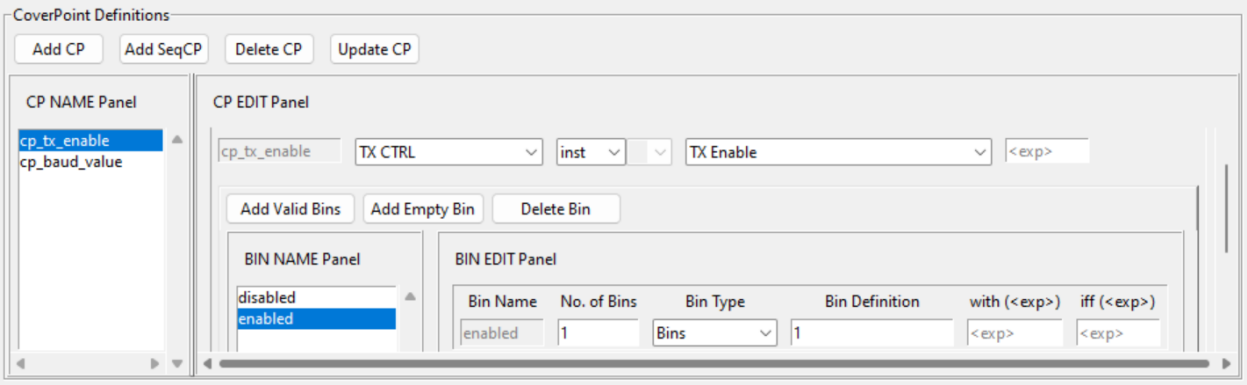

Figure 5. Example Coverpoint interface with added Coverpoints

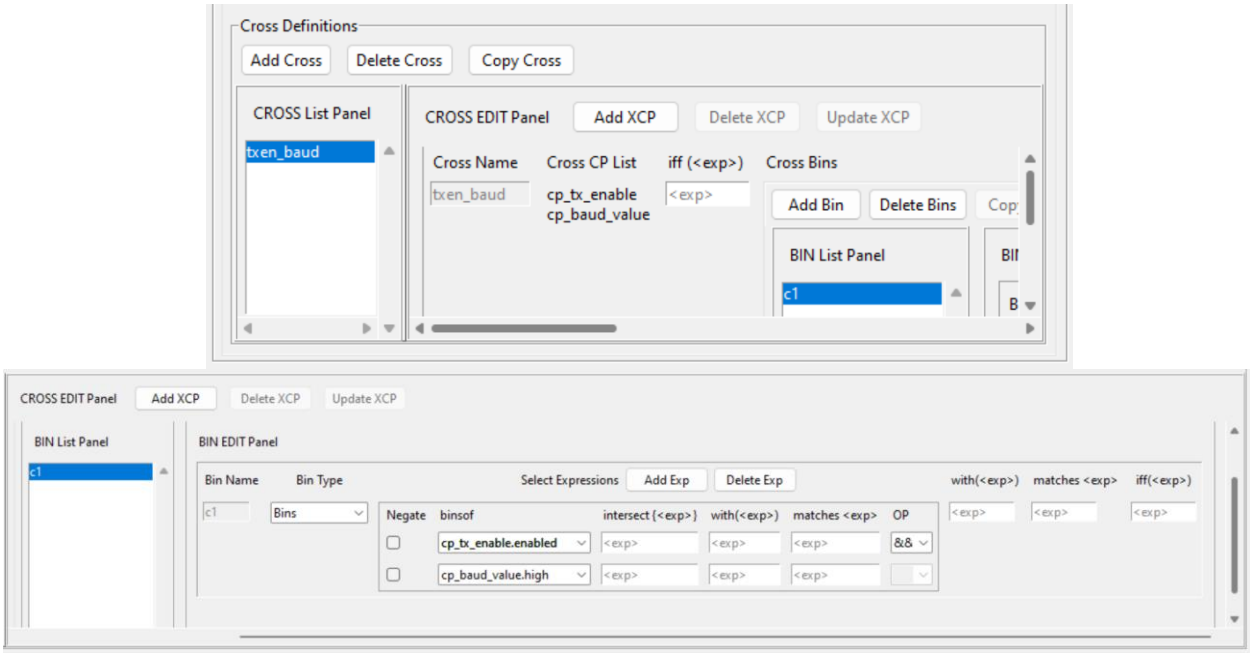

Figure 6. Example Cross Coverage Interface with Expanded Cross Edit Panel

```
covergroup CoverGroup_1(int inst);
    //Enabled option to track coverage for each instance of covergroup
    option.per_instance = 1;

    // Coverpoint for TX_CTRL Register/TX Enable Field
    cp_tx_enable: coverpoint ral_reg.tx_ctrl_reg[inst].tx_enable.value {
        bins disabled = {0};
        bins enabled  = {1};
    }

    // Coverpoint for BAUD Register/Baud Value Field
    cp_baud_value: coverpoint ral_reg.baud_reg[inst].baud_value.value {
        bins low    = {[0:9600]};
        bins medium = {[9601:115200]};
        bins high   = {[115201:$]};
    }

    //========= CROSS Definitions =========//
    txen_baud: cross cp_tx_enable, cp_baud_value
    {
        bins c1 = binsof (cp_tx_enable.enabled) && binsof (cp_baud_value.high);
    }
endgroup
```

Figure 7. Example CoverGroup defining Coverpoints and Cross Coverpoint

B. Sequence-based Coverpoint Case Study

During simulation, input stimulus is provided through RAL register transactions in the testbench. This sequence is captured and pushed into the input stimulus queue (input_txn_q) as part of the coverage block. The user provides a list of expected sequences to track using the GUI, as shown in Fig. 8.

Fig. 9 shows the sequence covergroup, sample1_sequence, which defines a coverpoint uart_seq_en. This coverpoint is triggered based on the return value of the check_uart_sequence function, specifically tracking the enabled bin associated with that return value. The check_uart_sequence function is responsible for generating exp_txn_q. It then calls the is_sequence_subset function at the sample point to check if exp_txn_q is a subset of input_txn_q. It is important to note that this coverage mechanism does not guarantee the functional success of the sequence; rather, it provides insight into whether the testbench has programmed the specified sequence. The compare algorithm in the is_sequence_subset function checks if a specific pattern of register transactions occurred in the correct order within the captured stimulus data. For each transaction in exp_txn_q, it compares the Action, Register, and Field names, as well as the Value. Since input_txn_q only contains complete 32-bit register writes, when exp_txn_q specifies a write transaction on a particular register field, a field mask is calculated and applied to the corresponding value in input_txn_q to identify a match.

To handle cases where a subset of transactions within a sequence may occur in any order, the methodology introduces dummy guard transactions at the boundaries of unordered sections within exp_txn_q, as specified by the user through the GUI. This allows the comparison algorithm to return a successful match if all relevant transactions are observed between the guard transactions in input_txn_q, regardless of their order and without intervening unrelated transactions.

This approach can be extended to define coverpoints for a wide range of complex test scenarios. Such scenarios are difficult to capture using simple transition coverpoints or even extensive cross coverpoint definitions, highlighting the flexibility and utility of the proposed sequence covergroup methodology. This approach is resource-efficient, as it performs queue comparisons only at sampling points, thereby obviating the need for additional threads to monitor transactions in real time. Furthermore, a database of exp_txn_qs is maintained in a machine-readable format, enabling easy loading, modification, and reuse by different users. The memory footprint remains minimal, as the exp_txn_qs are instantiated only within the scope of the comparison task, requiring storage of the input stimulus in a packed structure during simulation. Additionally, cross coverage can be defined on these sequence-based coverpoints to determine whether multiple sequences are observed before the sampling point. However, a limitation of this approach is that the temporal ordering of coverpoint hits cannot be determined solely through cross coverage; additional logic would be required to capture such ordering information.

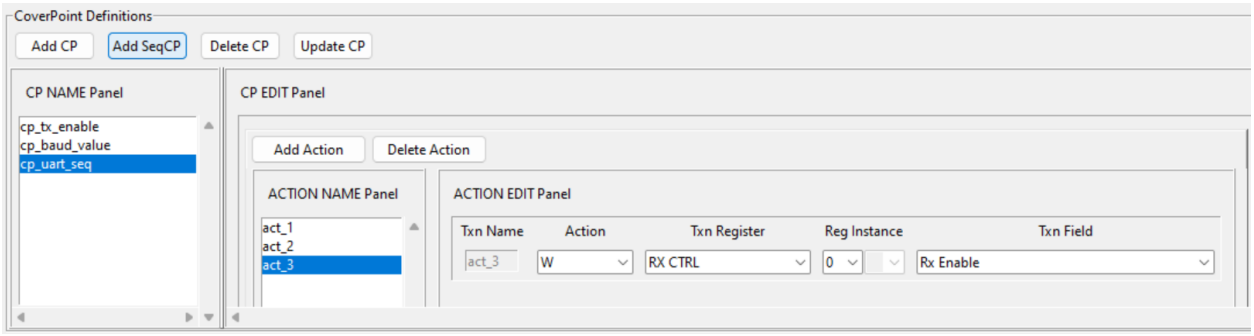

Figure 8. Example Sequence Coverpoint in Coverpoint Interface of CovDefTool

```
//Generated using GUI backend
covergroup sample1_sequence(int inst);
    // Track individual enable bits for each instance
    cp_uart_seq: coverpoint check_uart_sequence(inst) {  bins enabled = {1'b1}; }
endgroup

//Generated using GUI backend
function bit check_uart_sequence(int inst);
    reg_action_def exp_txn_q[$];

    exp_txn_q.push_back('{Action:WRITE_REG, Reg: ral_reg.baud_reg[inst],    Field: ral_reg.baud_reg[inst].baud_rate,    Value:16'h1c200});
    exp_txn_q.push_back('{Action:WRITE_REG, Reg: ral_reg.tx_ctrl_reg[inst], Field: null,                                Value:32'h3});
    exp_txn_q.push_back('{Action:WRITE_REG, Reg: ral_reg.rx_ctrl_reg[inst], Field: null,                                Value:32'h3});
    exp_txn_q.push_back('{Action:POLL_REG,  Reg: ral_reg.status_reg[inst],  Field: ral_reg.status_reg[inst].error,      Value:1'h0});
    exp_txn_q.push_back('{Action:WRITE_REG, Reg: ral_reg.tx_ctrl_reg[inst], Field: ral_reg.tx_ctrl_reg[inst].tx_enable, Value:1'h0});

    // Check if expected sequence is subset of input_stimulus
    return is_sequence_subset(exp_txn_q);
endfunction
```

Figure 9. Sample Sequence covergroup and function definition code

IV. Enhancements and Future Scope

The current coverage integration framework was developed to work efficiently within a specific UVM testbench, enabling precise sampling and targeted validation for IP-level designs. This focused implementation has demonstrated strong results in capturing meaningful coverage aligned with functional activity. As we scale toward SoC-level environments, which involve multiple IPs, hierarchical testbench structures, and shared verification components, the next step is to generalize the integration layer. By introducing abstraction mechanisms and configurable interfaces, the tool can adapt dynamically to diverse testbench architectures without requiring extensive customization. This evolution will enable seamless deployment across different SoC platforms, promoting reuse, scalability, and consistency in coverage-driven verification.

Register data can be loaded on demand to reduce initial load times and prevent GUI overload, especially in large SoC environments. Advanced search, tagging, and filtering capabilities will be introduced, allowing users to efficiently locate registers by attributes such as name, type, access mode, and coverage status. Registers can also be organized into logical groups, such as regfiles or feature sets, with collapsible views to streamline navigation and reduce visual clutter. To manage bin explosion, the tool supports the use of cross_auto_bin_max at the covergroup or coverpoint level, which automatically excludes all auto-generated crosses when at least one explicit cross is defined. Additionally, the GUI encourages the use of illegal and ignore bins, enabling users to exclude coverage bins that are irrelevant to the current verification scenario and thereby focus coverage analysis on meaningful results.

Assertions or dedicated monitors will be scoped to validate that sampling occurs only under the correct conditions. This would prevent false coverage hits due to incorrect or premature sampling.

The applicability to non-UVM-based environments is not explored in the scope of this paper but will be considered in the future.

V. Results and Summary

The proposed GUI framework demonstrates substantial improvements in coverage definition efficiency and accuracy for hardware verification workflows. The GUI tool was deployed across multiple verification engineers to develop comprehensive coverage models, including complex cross-coverage scenarios. The implementation generated a substantial codebase of 1,531 lines of coverage code, providing a robust dataset for performance evaluation. Comparative analysis between traditional manual coding approaches and the GUI-based methodology revealed remarkable efficiency improvements. The GUI approach achieved a 71% reduction in development time, requiring only 9 man-days compared to 32 man-days for equivalent manual implementation. This dramatic time reduction translates to significant resource optimization and faster project delivery cycles.

Beyond efficiency gains, the GUI framework demonstrated superior accuracy in coverage code generation. Error analysis showed an 83% reduction in defects, with only 36 errors identified in GUI-generated code compared to 210 errors in manually written code. This substantial improvement in code quality can be attributed to the framework's built-in validation mechanisms, standardized templates, and elimination of syntax-related human errors.

After rewriting a particular covergroup using the deployed tool, a previously undetected corner case bug was discovered. The process of refactoring the covergroup with the new tool revealed a gap in the existing test programming, which had previously allowed the bug to escape detection. Once this gap was identified and addressed, the underlying bug was exposed and subsequently fixed. This finding highlights the effectiveness of the tool in enhancing coverage definition and improving the thoroughness of verification, ultimately leading to the identification and resolution of subtle design issues that might otherwise remain unnoticed.

References

[1] IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language," in IEEE Std 1800-2023 (Revision of IEEE Std 1800-2017) , vol., no., pp.1-1354, 28 Feb. 2024, doi: 10.1109/IEEESTD.2024.10458102.