Tackling Missing Bins: Refining Functional Coverage in SystemVerilog for Deterministic Coverage Closure

Jikjoo Lee*, Tony Gladvin George*, Kihyun Park*, Dongkun An*, Wooseong Cheong*, ByungChul Yoo*
*Memory Division, Samsung Electronics
Email: (jjikjj.lee, tony.gg, kihyun0.park, dongkun.an, ws.cheong, byung.yoo)@samsung.com

***Abstract*- In the context of hardware design verification, defining functional coverage accurately in SystemVerilog remains a challenge, often due to human errors leading to "missing bins". This paper introduces a methodology aimed at enhancing functional coverage by identifying these overlooked bins. By treating coverage bins as a SystemVerilog queue and employing a "waiver function", this approach provides verification engineers a mechanism to efficiently determine whether sampled coverage bins are already accounted for in the coverage. Experimental validation, involving a cache managing IP, underscored the method's efficacy. The results revealed 14.1% of missing bins among the coverage holes using our proposed methodology, culminating in a 6.0% overall improvement in functional coverage. Thus, the proposed method not only rectifies human-induced inaccuracies but also improves the overall robustness of hardware verification.**

I. INTRODUCTION

In the context of functional coverage, despite achieving 100% functional coverage, bugs may still manifest. In such instances, verification engineers can find themselves perplexed, unable to pinpoint the oversights that led to these unexpected issues. Consequently, they are left with the disconcerting realization that they may not be able to confidently respond to the question, "Is verification truly complete?"

This paper addresses the inherent challenges tied to defining functional coverage in SystemVerilog, emphasizing the problems that arise due to human errors, notably the oversight of "missing bins". These are data bins that should ideally be part of the coverage but are unintentionally omitted. The core of this study revolves around proposing an efficient methodology to solve the recurring problem of defining functional coverage in SystemVerilog. The approach we propose is easily applicable due to its reliance on fundamental SystemVerilog syntax. This represents the methodology targeted towards achieving exhaustive coverage closure. The primary aim is to ensure a comprehensive approach to coverage, leaving no aspect unaddressed, thereby enhancing the overall quality of the verification process.

The rest of this paper is composed as follows: Section II introduces the challenges that arise when defining functional coverage and highlights the debugging difficulties encountered with conventional methods. Section III pertains to the relevant background and research. Section IV proposes a three-step method for composing functional coverage to address these challenges. Section V demonstrates the effectiveness of this approach when applied to real-world cases. Through the experiment, it is possible to ascertain the thoroughness of the validation by using the proposed methodology to confirm the potential outputs of the cache managing Intellectual Property (IP). Finally, it provides a conclusion and outlines potential future work.

II. PROBLEM STATEMENT

Illustrated in Figure 1 is the comprehensive set of coverage bins, each representing possible scenarios within a coverage point. This figure showcases the complexities surrounding the definition of coverage bins. Key sets are defined: the Universal Set (***U***), encompassing all potential coverage bins within a coverage point; the Ideal Set (***I***), representing the desired coverage; the Defined Set (***D***), the coverage bins explicitly identified by engineers for verification; and the Sampled Set (***S***), showcasing the coverage bins sampled during tests.

The coverage bins in the diagram are divided into five types, each of which has a specific meaning, as described in Table 1. "Covered bins" indicate that the item sampled by the test is covered by a coverage bin defined by the engineer. "Excluded bins" are coverage bins that are neither in the Ideal Set nor the Defined Set. "Uncovered bins" are those that have not yet been tested because of a lack of test scenarios.

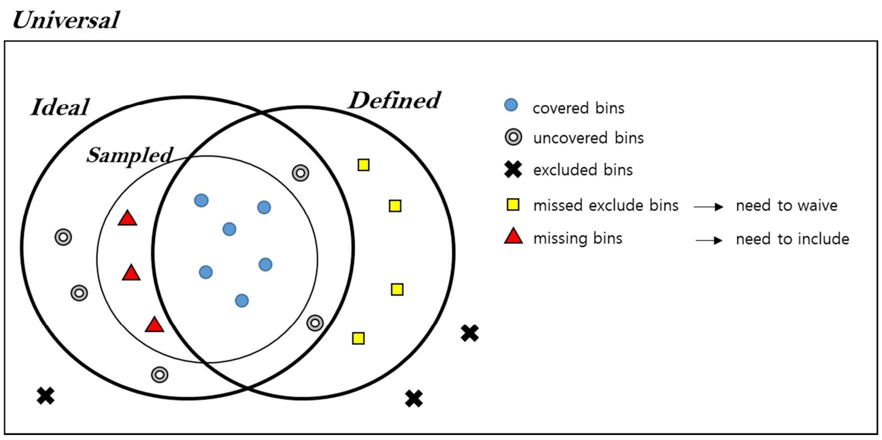

Figure 1. Coverage Points and Coverage Bins Diagram

Table 1. Meaning of the Five Coverage Bin Types

coverage bin type	set-builder notation	meaning
covered bins	$\{ x \mid x \in \boldsymbol{S} \cap \boldsymbol{D} \}$	sampled and defined
excluded bins	$\{ x \mid x \in (\boldsymbol{I} \cap \boldsymbol{D})^C \}$	excluded by ideal and defined both
uncovered bins	$\{ x \mid x \in \boldsymbol{S}^C \cap \boldsymbol{I} \}$	ideal but not sampled
missing bins	$\{ x \mid x \in \boldsymbol{S} \cap \boldsymbol{D}^C \}$	sampled but not defined
missed exclude bins	$\{ x \mid x \in \boldsymbol{I}^C \cap \boldsymbol{D} \}$	inadvertently included

The objective is to transition all coverage bins, barring the excluded ones, into covered bins. Ideally, a good alignment between the Ideal, Defined, and Sampled Sets is sought. This demands the engineer to meticulously adjust the Defined Set, ensuring the absence of "missing bins" and "missed exclude bins". "Missed exclude bins" emerge from human errors, representing ideally non-coverable bins mistakenly defined in the coverage point. Their presence prohibits reaching 100% coverage. On the other hand, "missing bins" refer to those bins that, despite being tested, aren't included in the coverage point due to oversights or over-exclusions. Their presence can falsely indicate comprehensive testing, leaving potential gaps in verification.

One traditional approach to address this is the "illegal_bins" keyword, which triggers notifications upon activation. However, its uncontrolled reporting poses challenges [1]. When solely utilizing the "illegal_bins" keyword, it becomes challenging to pinpoint the specific timing and rules under which illegal bins were triggered. In addition, it is necessary to use the Electronic Design Automation (EDA) tool to determine if you have hit any illegal bins. This complexity not only hinders debugging but also ultimately extends the time required to achieve coverage closure. In essence, there is a lack of a straightforward mechanism to prevent the reporting of errors in this context. Given that illegal bins manifest as errors, there exists a potential for test failure associated with their presence.

III. Background

In other research related to functional coverage, it has been suggested that validation is necessary to ensure all generated coverage bins are reachable. The Functional Coverage Management System (FCMS) [2] automatically converts table-based specifications into coverage. While the previous work has the advantage of being able to generate a SystemVerilog Assertion (SVA) model for formal

verification and a SystemVerilog functional coverage model for simulation verification, it requires the use of a separate tool called SpecGen and suffers from a performance drop of about 30%. The method we propose focuses on detecting "missing bins", and it can be easily and simply applied by just changing the way SystemVerilog functional coverage is written. When using our method, the performance degradation occurs within 3%, which is approximately ten times less compared to the conventional work.

Several EDA Tools provide features for the analysis of functional coverage, like merging coverage from multiple runs [3]. They also provide various insights from the perspective of analyzing cross-coverage. For instance, vManager has an "aggregate analysis" feature that analyzes the uncovered bins of cross-coverage, making it easy to understand the patterns of uncovered bins [3]. It also shows whether excluded coverage bins like "ignore_bins" and "illegal_bins" have been hit, along with their scores. However, it does not provide information on which test and at what time the excluded bins were hit. When verification engineers become aware of the existence of "missing bins", they need to check through which scenario that coverage bin was covered for accurate analysis. Our proposed method enhances the EDA Tool by incorporating a feature to analyze missing bins, which was not previously available. This new functionality provides detailed log information when missing bins occur. This saves the verification engineer's time in determining whether "missing bins" are reachable. Meanwhile, the EDA Tool remains the primary resource for analyzing uncovered bins.

IV. Proposed Solution

To achieve exhaustive coverage closure, we use a process of iteratively modifying the coverage model to find and correct differences from the ideal coverage. Figure 2 depicts the process of modifying coverage. Initially, the functional coverage is defined. The coverage model initially designated is highly likely to include holes, indicating the coverage that was inaccurately defined at the beginning. After conducting the test, the engineer analyzes the coverage results. When analyzing bins not covered in the coverage results, those deemed incapable of being covered are classified as "uncovered bins", while those assessed as coverable are labeled "missed exclude bins". If a bin is determined to be an "uncovered bin", we enhance the test scenario. If it's classified as a "missed exclude bin", we add the bin to the waive conditions. "Uncovered bins" can be easily detected using the traditional method of EDA Tools. The method we propose should be used in conjunction with the EDA tool approach, and it complements the existing methods.

Following the state-of-the-art methodology for functional coverage creation, engineers can review reports on "missing bins". If it is feasible to cover the reported "missing bins", they are removed from the conditions of the "waiver function". If coverage is indeed impossible, it necessitates an investigation into potential issues within the design or verification environment. If the coverage does not reach 100% or there are still missing bins, the process reverts to the run test stage, and the following steps are repeated. However, if the coverage becomes 100% and there are no longer any missing bins, the process terminates, and the coverage at that point is deemed the final coverage. Through this iterative process of refining coverage, tests are executed repeatedly to bolster coverage.

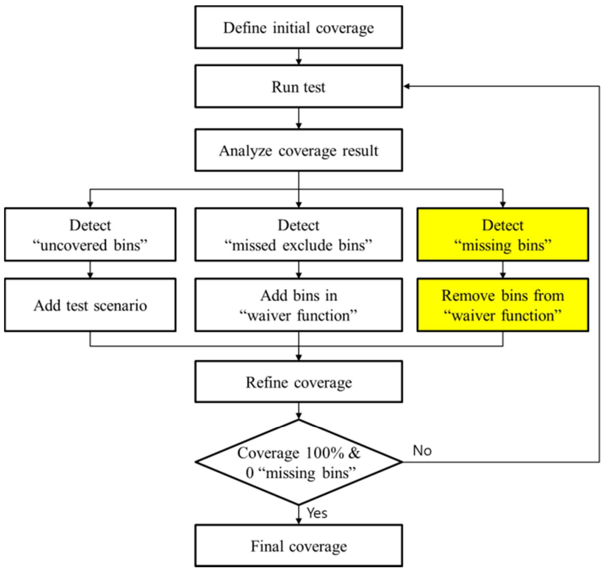

Figure 2. Process of Refining Coverage

This paper introduces a three-step approach to writing functional coverage in SystemVerilog. Through this methodology, engineers can effectively detect missing bins with ease. This approach purely leverages the coverage syntax of SystemVerilog. The relevant code snippets are provided in Figures 3, 4, and 5.

A. Defining Coverage bins in Queue

The first step involves defining the desired coverage bin within a queue, a specific data structure in SystemVerilog. The data pushed into this queue constitutes the coverage points for cross-coverage. The reason for defining coverage using the queue is to use "ignore_bins" flexibly and reactively in cross-coverage. As illustrated in Figure 3, a queue named StateCoverBin is defined with the capability to store 10-bit data. This is done to establish a cross-coverage that intersects 10 coverage points of one-bit. A critical component of this process is the "waiver function", determining if the data is reachable and ensuring only valid data is entered into the queue. This function, apart from verifying data, also offers reusability across multiple coverage points and can be utilized across various scopes or classes. The code snippet includes a "waiver function" named WaiveState. This function takes 10-bit data as input and provides an "IllegalIdx" as output. If, in the engineer's judgment, input data is feasible to be covered, the "IllegalIdx" is output as zero. If not, a value can be checked to see which rule includes it in the illegal condition. For all conceivable states, corresponding to the Universal Set, only those assessed as coverable, with a result of zero from the WaiveState function, are placed in the StateCoverBin. The queue now represents the Defined Set.

B. Defining Cross-Coverage Using CrossQueueType

The subsequent phase is about defining the cross-coverage using the CrossQueueType keyword. It is a SystemVerilog keyword utilized for the automatic definition of queues of tuples within each cross [5]. Our unique approach is to convert the queue used in step A to CrossQueueType to define the ignore bins in cross-coverage. This type of bin is advantageous for specifying bins for exclusion [4]. A unique function, createIgnoreBins, is defined to return a CrossQueueType. This function pushes all non-included instances from the queue, ensuring only the necessary coverage bins are retained in the cross-coverage. As depicted in Figure 4, we have defined ten coverage points from s9 to s0 as cross-coverage. The s9~s0 are coverage points created by dividing the sampled 10-bit data into ten bins, each consisting of 1 bit. The createIgnoreBins function incorporates into the output CrossQueueType only those states that are not present in the StateCoverBin. The keyword "ignore bins" is utilized for the cross-coverage bins that are the outcome of the createIgnoreBins function, to exclude them from coverage. This procedure allows for the transformation of states defined in the StateCoverBin into a cross-coverage bin format.

C. Reporting Missing Bins Pre-Sampling

The concluding step is about inspecting potential "missing bins" before sampling. The data intended for sampling is fed into the previously mentioned "waiver function". This process identifies if the data should be excluded. If the exclusion is deemed necessary, the return value from the "waiver function" exceeds zero, indicating the data pertains to the "missing bins". Such data is then reported via an error syntax, allowing verification engineers to quickly detect the presence of data from these "missing bins" in the logs. As demonstrated in Figure 5, the WaiveState function is utilized to procure "RespState", the data set for sampling, as an input and yielding "IllegalIdx" as an output. The implication of an "IllegalIdx" exceeding zero signifies a state where coverage is impossible. In such scenarios, an ERROR statement is employed to report the situation. The method of reporting is subject to modification depending on the configuration. This data's subsequent discovery facilitates an evaluation of the coverage definition's accuracy. Any discrepancies observed lead to modifications in the coverage definitions or in the test cases, thus refining the coverage reliability and enhancing verification quality.

```
  bit [10-1:0] m_anStateCoverBin[$];
  static function int WaiveState(int State);
    int nIllegalIdx = 0;
    casez (nState)
      'b?1????0??? : nIllegalIdx = 1    ;
      'b??????11?? : nIllegalIdx = 2    ;
      'b??????1?1? : nIllegalIdx = 3    ;
       ...
    endcasez
     return nIllegalIdx;
 endfunction
 ...
for(int nStatIdx=0; nStatIdx < (1<<10) ;nStatIdx++ ) begin
     if(WaiveState(nStatIdx) == 0 ) continue;
     this.m_anStateCoverBin.push_back(nStatIdx);
end
```

Figure 3. Defining Coverage Bins in Queue

```
cross_state :cross s9,s8,s7,s6,s5,s4,s3,s2,s1,s0 {
   function CrossQueueType createIgnoreBins();
       for(int nStat=0; nStat < 1<<10; nStat++) begin
           if(nStat inside {this.m_anStateCoverBin}) continue;
          else createIgnoreBins.push_back('{nStat[9], nStat[8], nStat[7], nStat[6], nStat[5],
nStat[4], nStat[3], nStat[2],nStat[1],nStat[0]});
      end
   endfunction

  ignore_bins ignore_cross = createIgnoreBins();
}
```

Figure 4. Defining Cross-Coverage Using CrossQueueType

```
nIllegalIdx = WaiveState(nRespState);
if(this.m_stConfig.m_nCovIllegalOn && nIllegalIdx > 0) begin
  `ERROR($sformatf("State RespState:%b Idx:%1d", nRespState, nIllegalIdx))
end
StateCovGrp.sample(nRespState);
```

Figure 5. Reporting Missing Bins Pre-Sampling Defining

V. Experimental Results

Experimental Setup

A series of tests were conducted to validate the effectiveness of the proposed method. These involved a Design Under Test (DUT) diagram showcasing the structure of the CacheManager (CM), a pivotal IP managing caches and their states. Figure 6 effectively illustrates the command flow of the CM. The CM incorporates modules referred to as StateMachines and several cache entries. These cache entries possess their 10-bit states, with each bit carrying a distinct meaning. For instance, the 0th bit could represent 'Latest', while the 1st bit may signify 'Dirty'. State transitions invariably occur due to the StateMachine.

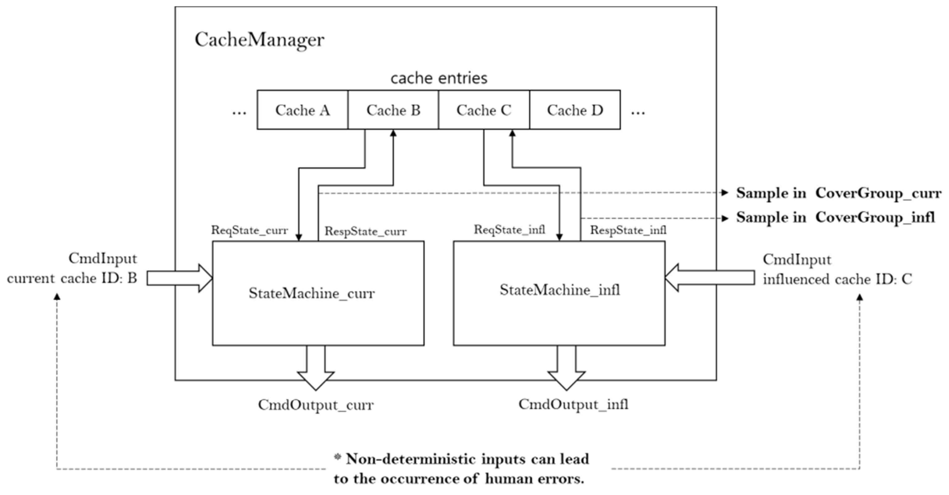

Figure 6. The Command Flow of the CacheManager

The StateMachine module is a type of deterministic finite state machine. There exist two types of StateMachines in CM: StateMachine_curr is utilized to modify the state of the cache currently being processed, while StateMachine_infl is used to alter the state of the neighboring cache influenced by the cache currently being processed. If the affected cache does not exist, StateMachine_infl will not work. The CM receives, as input, the index of the cache that needs to be currently processed (current cache ID) and the cache influenced by it (influenced cache ID), along with the command (CmdInput). The StateMachines receive the request state from the cache entry, determined from each respective cache ID, as ReqState_curr and ReqState_infl, accompanied by the CmdInput, serving as an input. A state-transition table is embedded within the StateMachine. This table describes information on how to modify the state based on the combination of command and request state, i.e., which bits to set or clear, and what output command (CmdOutput) to produce. The altered states, following the guidelines of the state-transition table, are outputted from the StateMachines as RespState_curr and RespState_infl. Subsequently, these states are stored as the state of the corresponding cache entries.

During the testing process, we implemented sampling each time an output of RespState was generated to confirm that every conceivable state had been transformed. The optimal strategy to examine the entire sample space of potential coverage (Ideal Set) is to predict every incoming command and its respective state. However, this approach encounters issues due to the following reasons: 1) Commands are received in a non-sequential or random order; 2) The Input Command is determined not only by the preceding Output Command but also by other external modules to the CM; 3) State transitions can occur in either the StateMachine_curr or StateMachine_infl. To predict the subsequent state, it is necessary to know the previous state as well as the upcoming Input Command. For the non-deterministic nature of such inputs, engineers are unable to define perfect coverage from the outset, which introduces the possibility of human error. Hence, we employed an approach of initially defining coverage for simple scenarios, then enhancing the coverage through repeated testing utilizing the method in Section III.

Results

Table 2 pertains to the results of our experiments. The experiment saw two cross-coverages, resulting in a total of 2048 states when all possible cases were examined. The study revealed complexities when isolating the Ideal Set ($\boldsymbol{I}$, a set of valid states) from the Universal Set ($\boldsymbol{U}$), given that the state-transition table elucidated bit transformations without detailing the resulting RespState. This complexity stems from the complex interplay between a myriad of input combinations and the ReqState.

For the sake of efficiency, the team initially outlined coverage for basic scenarios, named "Initially Defined Set" ($\boldsymbol{D}_{\text{init}}$), and then improved coverage iteratively based on tests. As a result, the "Initially Sampled Set" ($\boldsymbol{S}_{\text{init}}$) for RespState_curr and RespState_infl was found to be 202 in RespState_curr and 119 in RespState_infl out of a defined 304 and 296 bins, respectively. It implies that the initial coverage definition and test scenario only cover 53.5% of the total. After refining the coverage using our proposed method, we were able to achieve 100% coverage, with 228 coverage bins derived for RespState_curr and 165 for RespState_infl, which we have termed the "Final Defined Set" ($\boldsymbol{D}_{\text{final}}$). Having achieved 100% coverage and with no further missing bins emerging, we concluded that the "Final Defined Set" is identical to the "Final Sampled Set" ($\boldsymbol{S}_{\text{final}}$), the latter being the collection of samples obtained from the final test. Since the "Final Defined Set" and the "Final Sampled Set" are the same, we were able to designate them as the Ideal Set. Comparing the "Initially Defined Set" with the "Final Defined Set", there were 380 overlapping coverage bins, confirming that the "Initially Defined Set" has an accuracy of 63.3%, a figure derived by dividing the overlapping coverage bins by the total size of the "Initially Defined Set".

Table 2. The Size of Coverage Set

the size of set	U	D_{init}	S_{init}	$D_{final}=I=S_{final}$	$D_{init} \cap D_{final}$
RespState_curr	1024	304	202	228	200
RespState_infl	1024	296	119	165	180
Total	2048	600	321	393	380

* D_{init} accuracy : 63.3%

Throughout the process of improving the coverage using the method proposed in Section III, we tackled the uncovered bins by analyzing their properties to determine whether they fit within the Ideal Set. By tracking the transition of the previous states, the engineer could verify if the Input Commands were received in the correct sequence and determine whether that state could indeed be covered. If a bin was not covered but could be covered, test scenarios were constructed to cover these bins. If a bin was not covered and could not ideally be covered, they were earmarked on the "waiver function", effectively excluding them. By deploying the methodology proposed, "missing bins" - bins that should have been incorporated but weren't due to human error - were identified and the corresponding condition was removed from the "waiver function".

The results were encouraging, referring to Table 3. The paper's methodology assisted in efficiently uncovering 14.1% of bins that would have been overlooked using conventional techniques. Out of 255 total coverage holes, 14 and 22 "missing bins" were detected respectively, which represents a proportion of 14.1%. The remainder are "initially uncovered bins" that can be verified using traditional methods. There were 35 "uncovered bins" due to scenario deficiencies, and 184 "missed exclude bins" required exclusion.

Leveraging our proposed method arms verification engineers with a reliable strategy to identify and rectify "missing bins". Figure 7 illustrates the importance of eliminating missing bins in the process of enhancing the accuracy of functional coverage. Through our experiment, we were able to discover a total of 7 (Register Transfer Level) RTL bugs related to missing bins. Upon examining the results of the experiment, it was found that the method enabled the identification of a total of 36 "missing bins" out of the initially defined 600 coverage bins, equivalent to 6.0%. This indicates an enhancement of 6.0% in the overall quality of functional coverage, a significant improvement attributable to the implementation of our proposed method.

Table 3. Analyzing the Number of Coverage Bins with Holes

The number of coverage bins	Coverage holes	Initially uncovered	Uncovered	Missed exclude	Missing
RespState_curr	116	102	12	90	**14**
RespState_infl	139	117	23	94	**22**
Total	255	219	35	184	**36**

* The proportion of missing bins : **14.1%**

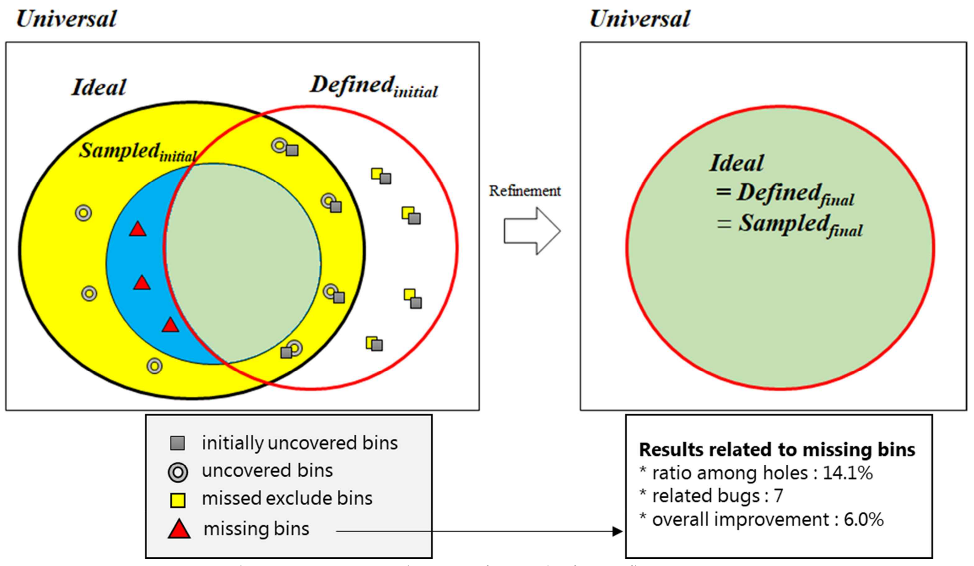

Figure 7. Coverage Bin Set Before and After Refinement

VI. Conclusion and Future Work

This paper introduces a methodology for writing functional coverage that effectively detects "missing bins" caused by human error. This methodology is motivated by the intent to confirm which functionalities of the design have been thoroughly verified. Despite achieving 100% coverage, the existence of these "missing bins" raises questions about the need for additional testing. To address this issue, we propose a methodology that defines a "waiver function" to describe the conditions of bins that cannot be covered and verifies whether a bin can be covered through the "waiver function" before sampling the coverage bin. This method, which exclusively utilizes SystemVerilog syntax, offers the advantage of being easy to use. Through the proposed method, engineers can identify and eliminate "missing bins", thereby enhancing the completeness of functional coverage. In the experiment, this methodology was used for the verification of the deterministic finite state machine, detecting a total of 36 "missing bins", thereby enhancing the overall verification completeness by 6.0%. The application of our proposed method has demonstrated effectiveness in addressing coverage points that demand thorough validation. For simpler coverage points, conventional methods can be consistently applied without encountering any issues.

Our proposed methodology is versatile, allowing for the sampling of not only 10-bit data but also of data in varying formats, including structured or object-oriented data. When defining transition bins to verify the command order, human error may occur during the functional coverage definition process. This methodology is capable of counteracting this by identifying "missing bins" during the transition bin definition.

A distinct advantage of utilizing the proposed method for functional coverage is its ability to detect previously overlooked "missing bins". Nevertheless, this process necessitates an engineer's intervention to validate whether the coverage bin corresponding to the "missing bins" resides in the Ideal Set. As a future development, we plan to research enabling reactive coverage closure through script automation. Through automation, we aim to diminish the necessity for human intervention in determining if a coverage bin belongs to the Ideal Set. This advancement aims to streamline the process and enhance the efficiency of coverage closure tasks.

REFERENCES

[1] "what is difference between ignore bins and illegal bins.", Verificaton Academy, last modified July 29, 2021, accessed Sep 1, 2023, https://verificationacademy.com/forums/systemverilog/what-difference-between-ignore-bins-and-illegal-bins.

[2] S.Ikram, J.Perveilier, I.Akkawi, J.Ellis, D.Asher, " Table-based Functional Coverage Management for SOC Protocols" in DVCon 2015, https://dvcon-proceedings.org/wp-content/uploads/table-based-functional-coverage-management-for-soc-protocols.pdf

[3] Cadence Incisive vManager User Guide, https://support.cadence.com/apex/ProductManuals?pageName=ProductManuals

[4] "How to Ignore Cross Coverage Bins Using Expressions in SystemVerilog", AMIQ Consulting, last modified September 17, 2014, accessed Sep 1, 2023, https://www.amiq.com/consulting/2014/09/17/how-to-ignore-cross-coverage-bins-using-expressions-insystemverilog/

[5] IEEE Standard for SystemVerilog,19.6.1.3, https://ieeexplore.ieee.org/document/8299595/metrics#metrics