Table-based Functional Coverage Management for SOC Protocols

Shahid Ikram, Jack Perveiler, Isam Akkawi, Jim Ellis, David Asher

Cavium, Inc., 600 Nickerson Road, Marlborough, MA 01752

***Abstract*-Functional coverage is a key indicator for the completeness of a SOC design, but SOC protocol coverage space is enormous and generating the valid coverage space is a daunting task. We are presenting an automated functional coverage management system (FCMS) for SOC protocols that takes advantage of table-based protocol specifications. The methodology has been used to create cover-points, protocol transitions coverage, and transaction coverage for multiple SOC chip design projects and has saved a large amount of engineering time.**

Categories and Subject Descriptors
C.2.2 [**Network Protocol**]: Protocol verification; B.3.2 [**Memory Structures**]: Design Aids—*Formal verification*; B.5.3 [**RTL Implementation**]: Design Aids—*Verification*

General Terms
Functional coverage, Verification

Keywords
Protocols modeling, Formal verification, Functional verification, Functional coverage

I. INTRODUCTION

Random simulation-based verification is the main vehicle for SOC verification and the key indicators for the quality and completion of this effort are coverage metrics. Coverage metrics can be categorized into two main classes [1]: 1) *Code coverage* and 2) *Functional coverage*. Code coverage exploits the structure of RTL to auto-generate coverage points and hence make sure that all parts of the implementation are exercised or visited by random stimuli. Most of the RTL simulator vendors provide built-in code coverage as well as state machine coverage generation tools. However, this auto-generated coverage information is never sufficient to show that all features of the design are exercised and hence designers and verifiers almost always have to add functional coverage [1][13].

Functional coverage captures the functionality of the design and provides an orthogonal view of random simulation quality by focusing on design features and not the implementation of the design (as is the case of code coverage). The creation of functional coverage models is mostly a manual process, where verifiers study high-level English specification of the design and work with designers to identify important design features. These design features are then translated into cover properties, cover points, cross-coverage [1] etc. to create *Coverage Monitors*, which run in parallel with the design during simulations and exerciser runs. The data generated by these *Coverage Monitors* is processed to estimate coverage achieved through random simulation. This is called *coverage analysis* [1][4][13].

The manual creation of the *Coverage Monitors* suffers from, 1) Outdated specifications, 2) Errors in manual translation, and 3) Management of a monumental number of coverage points. Consider the case of any industrial strength SOC protocol. The protocol has thousands of legally defined sequences, with many more illegal sequences. This observation yields two undesirable strategies for coding functional coverage points:

1. Manually write a coverage point for each of the thousands of legal transaction sequences, or
2. Cross cover all state variables and manually exclude the enormous number of illegal sequences.

Neither of these strategies works well for an evolving protocol and almost always leads to missing cases.

In this paper, we are proposing a table-based transaction-level functional coverage management system (FCMS) for SOC protocols. The high-level specification is described using extended state tables [8]. These tables are in ASCII format and are used to generate functional coverage collateral for various parts of the system. Formal and simulation engines are used to generate and analyze coverage data. The generated information is used to guide the design effort. The result is a coverage-based closed-loop system, which achieves faster design verification closure. These machine-generated coverage points reduce human error, drastically decrease the time required to code, and can rapidly incorporate changes to the protocol. The system is deployed in multiple SOC projects successfully with measurable gains in productivity. Finally, the proposed method can be applied to any protocol that can be described using extended state tables [8][15][16].

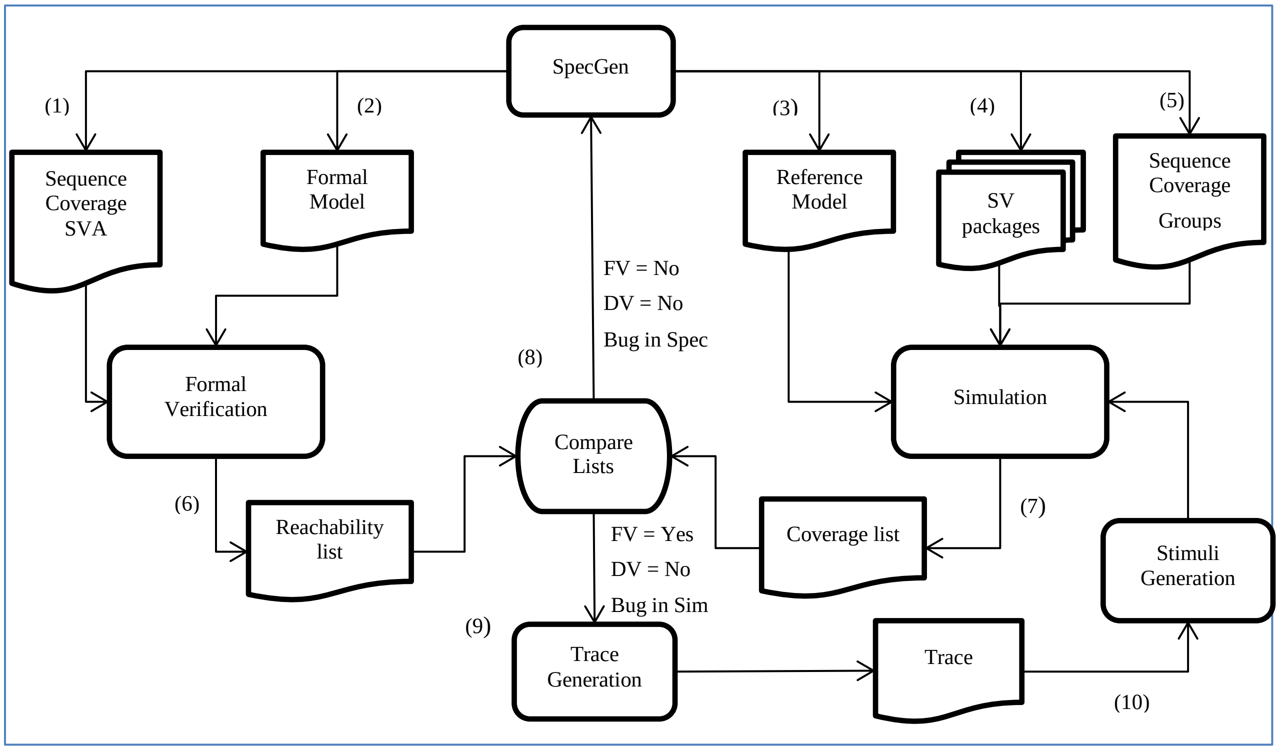

Figure 1: General Dataflow

II. A Functional Coverage Management System (FCMS)

We have implemented this system as a semi-automatic flow as described in Figure 1. The general dataflow of FCMS consists of four sub-flows:

1) Specification Generation (1,2,3,4,5)
2) Coverage Validation(6)
3) Coverage Data Collection(7) and
4) Coverage Analysis (8, 9, 10).

A brief elaboration of each of these components follows:

A. Specification Generation

This flow is based upon the tool called SpecGen (short for Specification Generator). SpecGen is a Perl-based tool, populated with valid states, initial states, commands etc. The setup of the tool is manual in terms of these basic behaviors but generation of tables is automated. The input to the tool is the architect's intentions i.e. protocol specification. The outputs of the tool are multiple specifications such as ASCII tables for visual purposes (not shown to avoid congestion), formal models, SV-packages for RTL implementation, reference models [15] for the verification environment and *coverage collateral* for coverage management flows. The general algorithm inside the specification generator is:

1. Start with a number of lists including:
    a. All commands that can transition states (for example requests, responses, probes, …)
    b. A state table, that is initially populated with all "Idle" states
    c. A table that covers all valid "commands", current "State" combination, and the resultant next "State"
2. Iterate over all the commands, and the state table, producing the next "State"
    a. If the new "State" is not in the table, it is added to the table
    b. The commands are recursively applied to new "States".
3. When the full table is generated, it is formatted and outputted in different formats, like RTL SV packages [1] for RTL etc.

The table used in step "1c" is effectively hand generated. It is a transition table that gives the next "State" for what the architect thinks the valid command/current "State" combinations are. The table also indicates what combinations of

command/current “State” are invalid (illegal). In step “2” a new table is instantiated, populated with the idle state only; all inputs are applied to it, and checked against the table from step “1c”. If the result yields a new next “State”, then that State is added to this new table, and all commands are applied to it. If the table in step “1c” indicates the combination as invalid, then these results are skipped, and iteration continues with the next command/state combination. If no entry is found, an error is indicated so the architect can fix it.
The arrows marked (1), (2), (3), (4) and (5) show the potential outputs of the specification generation flow. The flow (1) generates SVA [1] sequences and coverage properties. These coverage properties are formally verified against the Formal model generated from the flow (2). The correctness of the Formal model and any changes in the protocol (reflected through changes in tables) are validated using formal verification [15]. A formal verification tool generates two lists: 1) reachable covers properties and 2) unreachable cover properties. The unreachable properties show specification errors. The reachable property list is compared against the coverage list generated from simulations (flow (8)) as explained below. The flow (5) generates SV coverage groups. These coverage groups are used in conjunction with reference model (flow (3)) and SV packages (flow (4)) for simulation-based coverage generation.

B. Coverage Validation

The two key concerns about the auto-generated sequences are:

1) Are all the generated sequences reachable?
2) Did we miss any sequence?

The solution to the first question is straight forward. The coverage validation flow uses formal tools to validate that all the states, transitions and sequences generated by SpecGen are reachable. This validation is performed against a formal model that is also partially generated by SpecGen. The inputs to this flow are SVA coverage properties and a formal model and the output of this flow is a reachability list, describing what subset of coverage properties was reachable. Any unreachable coverage property has to be validated by the architect as it is a potential bug in the protocol or in the SpecGen tool.

The second question about a *possible miss* depends upon the correctness of our algorithm to find all the paths in a directed graph. It is a simple and well researched algorithm [14]. We did not find any misses through our cross-checking mechanisms (that were in place before FCMS).

C. Coverage Data Collection

RTL is the actual implementation of the design and must be fully covered. SpecGen auto-generates coverage points for each of the thousands of legal cases. These coverage points are used to model state coverage, transitions coverage, transaction/sequence coverage and cross-coverage for the protocol. System Verilog coverage groups are used to model these coverage entities in the reference model [15]. For RTL, SV-packages are annotated with coverage information that is used to flush the RTL coverage to text files during simulation. These are inputs to the Coverage Data Collection flow. The output of this flow is a composition of coverage data generated from RTL and the Reference model during simulations and is used to create a “reached coverage list”. The performance hit for these simulations was around 30% but coverage generating simulations were run only once every two weeks.

D. Coverage Analysis

The important precondition to the following coverage analysis is that all the coverage properties for all the flows have some method of back-annotation to show that they belong to the same source. Since we are generating them from the same source, this backward-annotation almost comes for free.

The coverage data generated from RTL and the reference model is used to analyze the coverage. This coverage is compared with the formal model’s coverage reachability list. There are four possible outcomes.

1. If a transaction is reachable in the formal model and unreachable in simulation, it represents a *hole* in the stimuli generation or a *bug* in RTL. The formal reachability engine is used to generate a guidance trace for simulation (flows 9, 10 in Figure 1).
2. If a transaction is reachable in both formal and simulation, we are done with it.
3. If a transaction is not reachable in formal but reachable in simulation, that is a bug in the formal model or a bug in the RTL/reference model.
4. If a transaction is reachable in neither formal nor simulation, then it is a bug in the specification and SpecGen needs to be fixed (flow 8 in Figure 1).

The coverage analysis step is the key to the efficiency of FCMS and contributes significantly to the convergence of the verification process.

III. Deploying FCMS

We have successfully deployed FCMS in multiple SOCs. In this section, we will describe the details of its application for a multichip interconnects protocol called OCI.

A. *O* Coherent Interconnect (OCI)*

OCI is a variant of directory-based cache-coherence protocols. OCI connects two to four O* III SOCs to appear as a single logical multicore processor (Figure 2). Coherency is implemented across Cores, Memory, Networks, I/O, and coprocessors. The architecture eliminates unnecessary memory copies.

At the top-level SOC, each address is mapped to a unique socket, which represents the address's home node. However, the address space is shared among nodes, and any other node may request data or ownership from any datum of the address space. If this datum falls in the requester's own domain, it will be a local request; otherwise it will be a remote request.

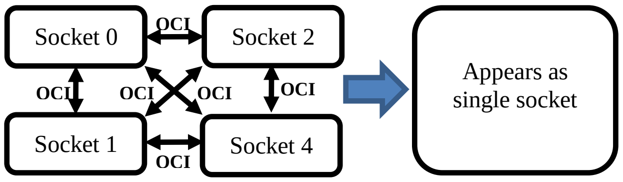

Figure 2: OCI Protocol

B. *OCI Specification Tables*

As expected, the full table contains a line for every combination of valid line state & input, providing the next state, and the OCI commands to issue to each of the nodes (the Output). For the Home Table, the state of the line consists of the "Cmd" which is representative of the request being processed, and the state of this node's cache directory (H for home), and the state of each of the other nodes' OCI directory (N1, N2, N3). Table 1 shows a sample of OCI specifications tables for a home node for a two-node setting.

TABLE 1

HOME

Current State			**Next State**			**Outputs**	**Inputs**
Cmd	**H**	**N1**	**Cmd**	**H**	**N1**	**N1**	**Req/Resp/Frwd**
none	E	I	none	S	S	RD_RSP (Read Response)	OCI_RD (Read from remote node)
none	S	S	LCL_WR	S	S->I	INV (Invalidate)	LCL_WR_LCL_A (local write to home address)
LCL_WR	S	S->I	none	M	I	-	INV_RSP (Invalidate response)

The Remote tables have a similar current/next state, but only one directory state "St" which is the state of its cache directory. Here the output consists of a command (request or response) that is sent to the home node, and a column for the possible response that can be also sent to the other remote nodes. Table size grows exponentially. For four nodes, we have around 10000 transitions but the whole process is automated. Simulations get a performance hit up to 30%. For the Formal part, we use a number of techniques for abstraction and speed, which are presented in [15] [16].

C. *Functional Coverage Transactions*

Generating protocol coverage points for legal states and single-step transitions is trivial. The interesting case is how the sequence/transaction coverage is generated.

The protocol contains *stable* as well as *transient* states [8]. A *stable* state does not have any outstanding command in process, whereas a *transient* state has an outstanding command. The stable states' set represents possible starting and end states of transactions. The transient states represent the intermediate states of the transactions. A transaction consists of a starting stable state and an ending stable state and with some possible transient states in the middle. A transaction may not have a transient state, for example in the case of a cache hit for an address, since no state transitions are necessary.

Algorithmically, we want to capture all paths that lead from a state with no outstanding transaction ("**Cmd** == *none*") to a state with no outstanding transaction, that is we want to capture all paths from the request arriving (or becoming active) to its exiting/retiring. Finding all paths between two nodes in a directed graph without any loops is an NP-complete problem. However, since our protocol graph is based upon independent command groups and can be divided into smaller graphs, the total time to construct these paths is not significant.

D. A Sample Home Transaction

We will use a small transaction to show the different stages of our flows. Table 2 shows a sample OCI home transaction. It is a "**Cmd**" going from "none" to "none" with one intermediate transition. In the initial state, remote node N1 has an exclusive (E) state for a given address. A core at the home node requests to share this datum (OCI_LD). OCI sends a forwarding request (FWDH) to the remote node N1. However at the same time, the remote node was in the process of evicting this address. Therefore the remote node replied with "I am invalid for this address" (REM_INV) message. OCI is an out-of-order protocol. Therefore, REM_INV may reach home before victim data (VDATA) reaches home. Meanwhile, REM_INV will put N1 in a transient state(S->I) in the home directory for this datum. These two messages may be received in opposite order, but that would be a different transaction.

TABLE 2

AN OCI HOME TRANSACTION

Current State			Next State			Outputs	Inputs
Cmd	**H**	**N1**	**Cmd**	**H**	**N1**	**N1**	**Req/Resp/Frwd**
none	I	E	E2S	S	S	FWDH	OCI_LD
E2S	S	S	E2S	S	S->I	none	REM_INV
E2S	S	S->I	none	M	I	-	VDATA

E. Modeling Coverage

Three different flows (3, 5, 1 in Figure 1), use coverage properties and cover groups generated from the same architectural specification. The ideal choice would have been SVA. SVA is an IEEE standard [11] and a powerful language to describe properties of a design in a declarative way with the ability to be used both in formal as well as simulation-based environment. However, we only used SVA for formal verification. We used *System Verilog Cover Groups* to model coverage properties in the reference model. The choice was made because our reference model is based upon System Verilog classes [1][11] and assertions cannot be instantiated in classes/objects in System Verilog. We also used annotated SV-packages for RTL coverage properties as it provided a much faster method for data collection as compared to SVA. Also, RTL holds much more detail about the design and timing the abstract SVA correctly was a challenge in itself.

F. SVA Properties

The total number of legal transactions generated from the protocol is in the thousands. However, we observed that most of the properties share similar templates of sequences. Therefore, we identified all the unique sequences from the transaction table and wrote a tool to generate a parameterized SVA sequence for each of these unique sequences. The parameterized SVA sequences were then instantiated for all the cases depicted in the original protocol tables [12].

Figure 3 shows one possible SVA sequence capturing the OCI transaction shown in Table 2. The general template observes four and tracks three states, 1) outstanding command for an address at OCI, 2) State of the address at the home node, 3) State of the remote node recorded at home. It also watches the input and output channels for the given address. The SVA sequence homeTrans_111 captures any protocol sequence resulting from a home core request while the datum is exclusive (E) at the remote node N1. We can identify two types of states in these sequences: waiting state and transition state. The waiting state models an arbitrary wait for something to happen. The transition state represents a protocol table line transition that should happen in one clock cycle. The operator "[*1:$]" lets the coverage monitor stay in a waiting state as long as the next state transition does not happen [12]. Once that transition happens, the monitor expects a one-cycle transition to the next waiting state.

Finally, cover_homeTrans_OCI_LD_1 creates an instance of a coverage monitor which, upon observing an OCI_LD request from a home core, starts following the sequence homeTrans_111. We generated SVA monitors for all the possible OCI transactions at the home node and remote nodes and ran them against our formal model. The total number of monitors was very large for formal verification, but each coverage monitor can be proved reachable individually. This observation provided sufficient parallelism to run chunks of the coverage monitors with formal verification without state-space explosion. In a week's time we were able to determine for all coverage monitors whether they were reachable or not.

```
sequence homeTrans_111(logic [A-1:0][3:0]Cmd, logic[A-1:0] [1:0]H_state,
logic[A-1:0] [1:0]N1_state,logic [A-1:0][4:0] output,logic [A-1:0][4:0] Inputs);
 (((Cmd ==  NONE)&&(H_state == I)&&(N1_state == E) && (Outputs == NONE )&& (Inputs == NONE ))[*1:$]) //Initial waiting state
##1  ((Cmd ==  E2S)&&(H_state == S)&&(N1_state == S) && (Outputs == FWDH)&& (Inputs == NONE ))//Transition sate.
##1  (((Cmd ==  E2S)&&(H_state == S)&&(N1_state == S) && (Outputs == NONE)&& (Inputs == NONE ))[*1:$]) //Waiting state
##1  ((Cmd ==  E2S)&&(H_state == S)&&(N1_state == S) && (Outputs == NONE)&& (Inputs == REM_INV )) //Transition state
##1  (((Cmd ==  E2S)&&(H_state == S)&&(N1_state == S_I) && (Outputs == NONE)&& (Inputs == NONE ))[*1:$]) //Waiting state
##1  ((Cmd ==  E2S)&&(H_state == S)&&(N1_state == S_I) && (Outputs == NONE)&& (Inputs == VDATA )) //Transition state
##1  ((Cmd ==  E2S)&&(H_state == S)&&(N1_state == I) && (Outputs == NONE)&& (Inputs == NONE )); //End state
endsequence
//An instance of transaction coverage for OCI_LD request from a core.
cover_homeTrans_OCI_LD_1:cover property((Request == OCI_LD) ##1 homeTrans_111(cmd[addr],h_state[addr],n1_state[addr],output[addr],input[addr]));
```

Figure 3: SVA model of a home transaction

G. *System Verilog Cover Bins*

We used System Verilog coverage constructs like coverage groups, cover-points, and cross-coverage to model coverage monitors for the reference model of OCI being used in the verification environment. The reasons for this choice are explained above in section *Modeling Coverage*. System Verilog [1][11] provides constructs to model transitions as coverage bins. A state transition can be modeled as *A=>B*, where *A* and *B* can be a collection of signals concatenated together. Furthermore, there are operators available for consecutive repetitions (*A[*5]=>B*) as well as non-consecutive (*A[=5]=>B*) repetitions. These constructs and operators provide sufficient language support to clone SVA properties used in the formal model as *coverage bins* inside the cover groups defined in the reference model.

Figure 4 shows an implementation of the coverage monitor for the home table's protocol. The coverage monitor is defined as a System Verilog *covergroup* construct [1][11] and is part of the class defining an address protocol behavior in a verification environment. *Request_type* coverpoints are used to group various core requests to the protocol. The transitions are modeled as the aggregation of signals representing protocol state. The actual transitions are auto generated (file "HOME_TRANS_COV.SV"). There are thousands of these transitions and a sample for our running example case is shown at the bottom of Figure 4. SDELAY is a system parameter that defines the arbitrary length of the time a monitor may wait in a state and depends upon the implementation. In our case, a value of 10 for SDELAY was sufficient.

Once created, coverage bin *HomeTrans_OCI_LD_1* will trigger for an address when the remote node has exclusive state on this address and a core from the home issued *OCI_LD*. It will watch for a forwarding message put forth for the remote node and then wait for the response to appear. The behavior is essentially the same as of the sequence in Table 2 and *cover_homeTrans_OCI_LD_1* in Figure 3.

```
covergroup Home_Monitor;
requests: coverpoint  request_type {
bins cov_LD[] = {OCI_LD};
....//Other core requests and their bins.}
transitions: coverpoint {request_type,cmd,h_state,r_state,output,input }
     {//Auto generated transitions cover points.
    `include "HOME_TRANS_COV.sv"         }
  endgroup //Home_Monitor
                    ....... A sample auto-generated coverpoint from HOME_TRANS_COV.sv.....
bins HomeTrans_OCI_LD_1 = ({OCI_LD,NONE,INV,EXL,NONE,NONE}  => {OCI_LD,E2S,SHR,SHR,FWDE,NONE }=>{OCI_LD,E2S,SHR,SHR,NONE,NONE } )
[*1:SDELAY]=> {OCI_LD,E2S,SHR,SHR,INVL,NONE }=>{OCI_LD,E2S,SHR,S_I,NONE,NONE } ) [*1:SDELAY]=> {OCI_LD,E2S,SHR,S_I,VDATA,NONE
}=>{NONE,NONE,SHR,INV,NONE,NONE } ) ;
```

Figure 4: A Coverage Monitor

H. *Annotated RTL SV-Packages*

The previous three sections described our coverage models at abstract levels. The reference model coverage described in the last section collects data at cycle-level. The RTL design has many more details in it and must be covered in depth as that is what matters most. As mentioned earlier, part of the RTL is auto-generated from Protocol tables as SV-packages. We enhanced these packages with annotations such that each table-transition has a unique identification label. During each simulation, for each executed transition, its unique label is pushed into special instrumentation buffers (these buffers are not synthesized). When a protocol cycle completes, the sequence it corresponds to is recorded as having occurred. At the end of each simulation-run, these sequence counts are dumped into text-files. A script is run on the text-files, looking at the transition labels and their sequences, and compares it to the sequence table generated from the protocol (and used in reference model and formal model coverage generation) and accumulates coverage data. The output of the script provides continuous updates on the covered sequences in exercisers' runs. Additionally, sequences that are not possible solely because of stimulus restrictions are filtered so that we can determine the true coverage of the simulation exercisers.

I. *Results*

Four types of functional coverage were generated and analyzed during this work, 1) Legal state coverage, 2) Transition coverage, 3) Transaction coverage, and 4) Cross coverage. *Legal state coverage* and *Transition coverage* were the main source of finding bugs/coverage holes during the evolutionary part of the protocol and found a number of architectural bugs using formal tools. Formal tools did find a few unreachable transactions as well and led to the fixes in the protocol. As the protocol matured and RTL evolved, *Transaction coverage* and *Cross coverage* became major concerns.

Table 3 provides an abbreviated bug report of our work. Chip-1 and Chip-2 have different cores and hence different instruction sets. The OCI protocol was changed by approximately 20% between Chip-1 and Chip-2. We did not use formal for RTL and hence the "0" entries in RTL/formal cells. Architectural simulation was done using Spin [10] and found a few bugs and holes in the early explorations of the protocol. The protocol matured between chips and therefore the number of bugs captured and holes discovered reduced significantly but the strength of FCMS lies in having a proof that we covered all the legal space.

TABLE 3

OCI BUG REPORT

		Architectural Spec		RTL	
	Tool	Bugs	Holes	Bugs	Holes
Chip-1	Formal	15	20	0	0
	Simulation	2	3	45	20
Chip-2	Formal	5	7	0	0
	Simulation	0	0	35	8

The ROI (return in investment) of FCMS is the time saved in reaching our goals of tape-out in record time (savings of many man-months) as it helped us to quickly converge on only true failures/misses by weeding out any false failures/misses.

IV. Related Work

Previous research [3][5][6] has looked at ways to accelerate generation of functional coverage. Man-Yun Su etc.[3] devised a new language to capture design features for coverage purposes. It provides a succinct mechanism, but still the process is mostly manual and design iterations at the architectural-level have to be repeated through coverage models as well. Youn-Su Kwon etc.[5] work focuses on using HiTER as a specification language and uses it to generate temporal events for coverage. Shireesh Verma etc. [6] use static analysis of the behavioral Verilog to auto-generate cover-groups. We use standard state tables [8] as input specification and validate generated coverage properties. It is also a very different approach from smart simulation [9].

Our approach uses both formal and verification engines. However it takes advantage of these techniques at architectural-specification-level, unlike smart simulation [9] which enhances design exploration capabilities during simulation runs. Finally, our coverage analysis techniques takes leads from [2] [4], but again our emphasis is on facilitating a quick turn-around loop for protocol changes and pin-points any new bugs/holes caused by it.

V. Conclusions

We have presented a functional coverage management system (FCMS) for SOC protocols. Our generation of coverage properties from protocol extended state-tables specification for simulation as well as formal tools provided us multiple advantages over other approaches[3][5][6]. The input tables are readable and easily manageable. The protocol changes are incorporated and validated via thorough analysis and become part of regressions right away with little intervention. Therefore, the validator and designer can focus on coverage holes and bugs in the design instead of worrying about bringing the changes at the architectural-level to the verification environment and weeding out false errors. Moreover, our system contributes to generating more interesting functional coverage cases like *Event cross-coverage* across home and remote FSMs, *Transition coverage* of all the legal transitions, and *Transaction Coverage* of all legal transactions as a sequence of legal transitions. Our approach finds coverage holes and design bugs at the architectural-level as well as RTL-level. The key outcome of this system is a faster convergence of design effort and time savings. We have applied it to multiple SOC efforts and saved months of time to achieve tape-outs. The work was part of a bigger effort for collaboration of formal and dynamic verification across different design phases to facilitate faster convergence of SOC designs [15][16].

VI. Acknowledgements

We are thankful to our team for their support through all this work with special thanks to Wendi Missagia for technical editing as well as formatting help.

References

[1] Spear, C. and Tumbush, G. 2012. *System Verilog for Verification*. Third Edition. Springer, ISBN 978-1-4614-0714-0, 608 pages, 2012. . DOI: http://dx.doi.org/10.1007/978-1-4614-0715-7

[2] Tasiran, S. Yu, Yuan and Batson, Brannon. 2003. Using a Formal Specification and a Model Checker to Monitor and Direct Simulation. In *Proceedings of DAC2003, June 2-6, 2003.* DOI= http://doi.acm.org/10.1145/775832.775926

[3] Su, Man-Yun, etc. 2006. FSM-based Transaction-level Functional Coverage for Interface Compliance Verification. In *Proceedings of Asian and South Pacific Conference on Design Automation, Jan 24-27, 2006.* DOI: http://dx.doi.org/10.1109/ASPDAC.2006.1594726

[4] Lachish, O., etc. 2002. Hole Analysis for Functional Coverage Data. In *Proceedings of DAC2002, June 2-6, 2003.* DOI: http://dx.doi.org/10.1109/DAC.2002.1012733

[5] Kwon, Young-Su, etc. 2004. Systematic Functional Coverage Metric Synthesis from Hierarchical Temporal Event Relation Graph. In *Proceedings of DAC2004, June 2-6, 2003.* DOI: http://dx.doi.org/10.1145/996566.996580

[6] Verma, Shireesh, Harris, Ian etc. 2007. Automatic Generation of Functional Coverage Models from Behavioral Verilog Descriptions. In Proceeding of DATE2007, April 16-20. DOI=http://doi.acm.org/10.1109/DATE.2007.364407

[7] Weber, Ross. 2011. Modeling and Verifying Cache-Coherent Protocols, VIP, and Designs. *Jasper Design Automation, June 2011.*

[8] Sorin, D.J. etc. 2002. Specifying and Verifying a Broadcast and a Multicast Snooping Cache Coherence Protocol. Reasoning about naming systems. *Transactions on Parallel and Distributed Systems (TPDS) vol. 13, number 6, June 2002.* DOI: http://dx.doi.org/10.1109/TPDS.2002.1011412

[9] Ho, Pei, etc. 2000. Smart Simulation using Collaborative Formal and Simulation Engines. In *IEEE International conference on CAD*, pages 120-126, 2000.

[10] Holzmann, G.J. 2004. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, ISBN 0-321-22862-6, 608 pages, 2004.

[11] IEEE 1800-2012. *IEEE Standard for SystemVerilog—Unified Hardware Design, Specification and Verification Language.* 2012.

[12] Vijayaraghavan,S. etc. 2005. *A Practical Guide for system Verilog Assertions.* Springer, ISBN 0-387-26049-8 334 pages, 2005.

[13] Bergeron, Janick. 2006. *Writing Testbenches using System Verilog*. First Edition. Springer, ISBN 978-1-4419-39784-0, 406 pages, 2006. DOI: http://dx.doi.org/10.1007/0-387-31275-7

[14] Sedgewick, Robert and Wayne, Kevin. 2011. *Algorithm*. Fourth Edition. Springer, ISBN 978-0-3215-7351-3, 992 pages, 2011.

[15] Ikram, Shahid and Akkawi, Isam, etc. 2013. Modeling and Verifying a Coherence Protocol using JG-ARCH. In Proceedings of Jasper Architectural Formal Verification Forum 2013, Santa Clara, CA, Oct. 2013. DOI: http://dx.doi.org/10.13140/2.1.1047.4245

[16] Ikram, Shahid and Akkawi, Isam, etc. 2014. A Framework for Modeling, Specifying, Implementation and verification of SOC Protocols. In IEEE SOCC Conference Proceedings 2014, Las Vegas, NV, September. 2014. DOI: http://dx.doi.org/10.13140/2.1.4094.8480