FORMAL ARCHITECUTRAL SPECIFICATION AND VERIFICATION OF A COMPLEX SOC

Shahid Ikram Isam Akkawi David Asher Jim Ellis
{Shahid.Ikram, Isam.Akkawi, David.Asher, Jim.Ellis}@cavium.com
Cavium Networks, 600 Nickerson Road, Marlborough, MA 01752

Introduction: We are presenting a formal verification effort of a complex SOC architectural and microarchitectural specification. We started with an architectural specification in a tabular format. These architectural tables define high-level design behavior using finite state machines models. These tables also contain information for the microarchitecture actions. We first identified the different roles that can be played by the different components of the architecture and divided the architectural tables into sub-tables according to this analysis. We then identified the messages passed around among the components and defined the message channels accordingly. Finally, we figured out microarchitecture steps needed to implement each role for each sub-table transition. We created a verifiable, formal verification environment based upon these findings with the following challenges: 1) ensure that all the tables' transitions are reachable, 2) ensure that all design specific properties were passing, and 3) ensure that all the functional sequences were executable. Tens of architectural and microarchitecture bugs were found and fixed.

This work builds upon earlier reported works [2, 3, 4]. The innovations reported in this paper are:

1) Implementation of the micro-steps for verification of the microarchitecture.
2) Implementation of the macro-steps for verification of the multicycle operations.
3) Automatic stimuli generation.
4) Sequence coverage simplification and debug acceleration using unique identifiers.

Architecture: Figure 1 shows an overview of the architecture. The system can be configured to work as a one-chip or a two-chip system. In the two-chip system, a single system illusion is created through a complex proprietary architectural protocol. Each chip consists of a set of ARM cores and company-specific IPs and maintains its own memory. Furthermore, each chip may have multiple IOB bridges, accessing data from the memory space of any of the chips. An IOB bridge may facilitate multiple I/O devices through the IOB protocol. The memory agents are the memory-request handling components which provide interfaces to the system memory. The AP agents are interfaces to the CPU cores and handle read/write requests generated by CPUs. The interconnection fiber emulates multidirectional channels. The protocol presents an interesting challenge to formal verification. The protocol captures the interaction of AP agents, memory agents, IOB agents, and an interconnection fiber. The protocol was defined on per instruction basis. This feature facilitated the generation of a subset of protocol by selecting a subset of instructions. That was a big help in formal verification where state space explosion is always a challenge.

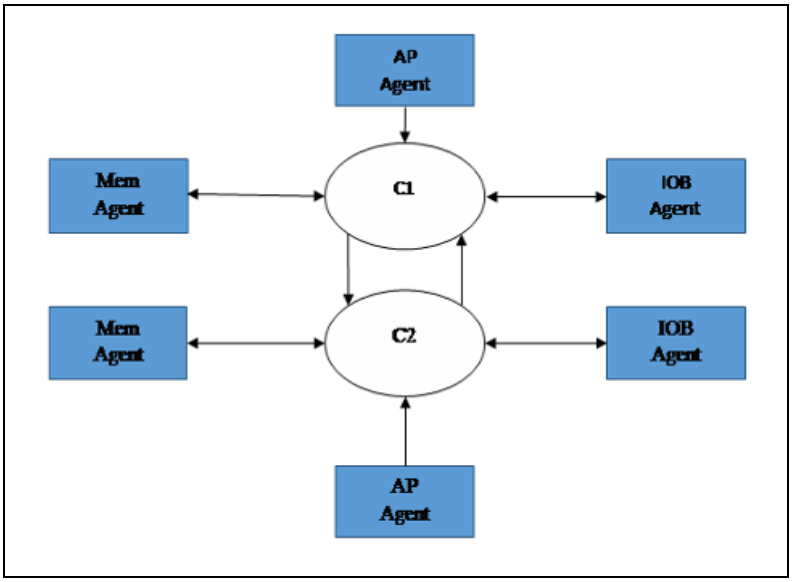

Figure 1: The SOC Architecture

Table 1: Transient states in a Protocol Table

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

Formal Specification: The core of our effort was based on an extended Table-based formal specification [1] [2]. The table generation was partially automated. We used Perl scripts to facilitate the table generation. The helper routines were used to enlist generic rules, and to provide the default states (i.e. only need to enter the changing bits/states for that state). However, most of the cases were fairly unique and hence a significant amount of the work in table generation was manual. The amount of the manual effort implied introduction of many inadvertent errors in the specification. Therefore, a formal verification was needed to weed out these potential oversights as well as to validate the global interaction of the architectural components.

A sample of a hypothetical table for the home node is shown in Table 1. The idea is to extend the protocol tables in such a way that all the *transient states* are also described explicitly as shown in the third row of Table 1. A *transient state* represents status of a component while it is in the middle of a transaction. It results in much larger protocol tables, but at the same time eliminates all the guess work from the specification. We verified these tables directly using formal verification and then used the same tables verbatim in the RTL, in the design verification environment as part of a predictor, and in the coverage collection as transition coverage.

The generation of these formal specification tables depends upon a number of steps. The effort requires the partition of the protocol in a systematic way and therefore needs a deeper understanding of the protocol. Next, we describe these steps in detail.

Agents and messages: The first challenge was to identify the actors or agents involved in the architectural specification. The second set of items was to figure out what is the minimum vocabulary the actors need to communicate with each other. The rest of the effort built upon this analysis. In our particular architecture, the protocol was built around the notion of maintaining consistent valid states. As mentioned earlier, the protocol was defined on per instruction basis. For a given instruction, there was a home agent and a probable remote agent. Also, there were local and remote IOB agents that could be accessed by this instruction. Finally, there was a probable memory agent for the instruction.

Next, we need to see the kind of messages these agents pass around. There will be request messages like a local AP agent or remote AP agent or an IOB agent requesting a datum. The servicing agent (the home agent of the datum) may respond with an acknowledge message as well as a data carrying message. Moreover, a servicing agent, while in the middle of a transaction (not a transition, as transitions are atomic), may receive a second request from a different agent and hence may need to generate a forwarding request. Therefore, we need at least four types of channels, one for each of the message types i.e. request channels, response/acknowledge channels, data channels, and forwarding channels. These channels must be independent to avoid deadlocks. We have found a deadlock in one of our earlier efforts because RTL decided to use a common pool of the registers for the different channels [2].

Each agent also has to maintain its current state. For instance, if an agent is already processing a request, it may be in a transient state like "S->I" in the last row of Table 1. Finally, there is a set of configurations that defines the behavior of an agent. For instance, we may configure the home agent for a write-back policy or a write-through policy. We include a dirty bits indicator, a replacement cycle indicator etc., in this set. The set also has a fault bit to model a random address fault.

We are sharing an abstracted snapshot of the input columns of the protocol in Table 2. We process only one type of message at one time i.e. if there is more than one message available from the different channels at the same time, we randomly pick one of the available messages. Please note, there is no correlation between the rows shown in this table. The entries are chosen to show samples of different types of commands. The first row shows a 'read' request from the remote node. The second row shows a forwarding message from the remote node.

Table 2: Input columns of a Protocol Table

Current State			New Request Cmd			New Ack Cmd		New Data Cmd	
Transient State	**Home**	**Remote**	**Requester**	**Server**	**Cmd**	**Requester**	**Cmd**	**Requester**	**Cmd**
NONE	I	I	Local	Local	Read	none	none	none	none
WI00	I	K	Remote	Local	For-ward	none	none	none	none
CSFH	S->I	K->E	none	none	none	Remote	PRE	none	none
ED00	I	D->E	none	none	none	none	none	Remote	PRD

The forwarding messages and request messages share the same columns in the table but represent different channels. The third row shows an acknowledge message from "Remote" as it is moving from a pending state "K" to exclusive state "E". The last row shows reception of a data command from Remote to Home. The inputs column set also includes configuration fields. The total number of columns in the input part of the protocol tables was around fifty. The total number of rows in the tables was around 6000. We also enumerated the total number of states of each agent. The total number of states of the agents including the transient states was around 500.

Roles Identification: The agents involved in the architectural level behaviors are essentially finite state machines. Ideally, we can use one home table and one remote table for the home agent and the remote agent respectively. Let us assume each of them contains 300 states. It means the composite finite state machine modeling the protocol will have a possible 90000 states. That is generally too big to be handled by a formal tool. Luckily, we can divide and conquer this problem.

Please notice that the agents exhibit different roles in the different states of a given datum. These states can be grouped into functional sets called roles, for instance, how a Home will behave to a new request from IOB when it already has an outstanding request from the remote node. This can be best captured in a forwarding role that includes all the states of the home node, where it may have to handle forwarding cases.

A home agent may be serving a local request or responding to a remote request or handling a forward. If Home is handling a forward, it cannot process a new local request. Similarly, if Home is responding to a remote request, it cannot handle a forward at the same time. On the other end, if the remote agent is processing a remote request, it will not process a forwarding request at the same time. These observations allowed us to divide home, remote, and IOB tables into smaller role tables. Depending upon the state of the agent and the type of the incoming message, each of the agents can play exactly one role at given time. Each transition is atomic, and messages per channel are delivered in order. However, we chose randomly among all the available messages from the different channels. Therefore, we covered all the possible scenarios of the protocol messages' ordering with respect to the message types.

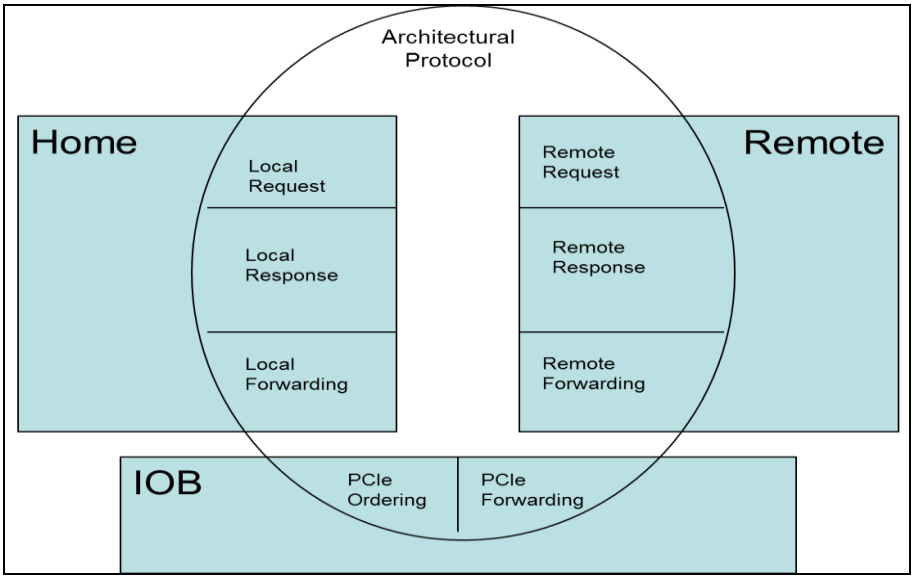

Figure 2: Architectural Agents and Roles

Table 3: Microarchitecture Steps

R0..R7	Mem Reads	Mem Writes	Comment
000xx	0x000	W00S0	R0..R7->T, T->Bus, Bus-> (MDR,MAR) MDR-> Memory
I00xx	0x000	0V000	IR -> Bus, Bus->MAR, MAR->Memory,…
0W0xV	0x000	0V0S0	R0..R7->ALU,…,(MAR,MDR)->Memory)
0W0xF	RF000	00000	Memory->MDR, MDR->Bus, Bus->R0..R7

A generic roles' description for the protocol is shown in Figure 2. Again, we are presenting an abstract view of the architectural agents and their roles. The actual implementation has had around 20 roles in total. Also, IOB tables had far fewer columns and were modeled in a different manner. We defined four role statemachines, one for home, one for remote, one for memory agents, and one for IOB agents. They captured different roles of these agents as states and legal transtions between these states.

Micro-steps: The enhanced protocol tables with all the transient states present an architectural view. Each transition in these tables represents the execution of an architectural step. Generally, in any design, an architectural transition represents a number of microarchitecture transitions (or steps). If a microarchitecture has been realized, we can further enhance architectural tables with microarchitecture transitions/steps and investigate their correctness.

Let us assume a generic computer microarchitecture shown in Figure 3. R0..R7 is the register file. IR is the instruction register, MAR is the memory address register, MDR is the memory data register, and PC is the program counter. There is also a control unit called CU and a memory system. These microarchitecture components are connected through a bus as well as in some cases through direct connections.

We have enhanced our protocol tables with additional columns representing microarchitecture transitions/steps. A sample table is shown in Table 3. The first row in this table shows the implementation of an instruction that processes the data present in one of the registers in the register file. ALU processes the data and puts it in its output buffer T. The data is transferred from T to MDR through the bus. Finally, data is written back to the memory from MDR using the address present in MAR. The last row in the table shows the micro-operations involved in loading the data from memory to the register file. In this case, the data will be transferred from memory to MDR, from MDR to Bus and finally from Bus to the register file R0..R7.

Note that it is important to enforce order among the micro-steps so that the intended control and data flows do not breakdown. Unfortunately, this ordering cannot be captured in the architectural tables as these tables only provide information on which micro-steps are involved in each architectural transition and nothing about their order of execution. An additional set of tables is needed to implement this ordering requirement. These tables model the microarchitecture-level understanding of the design.

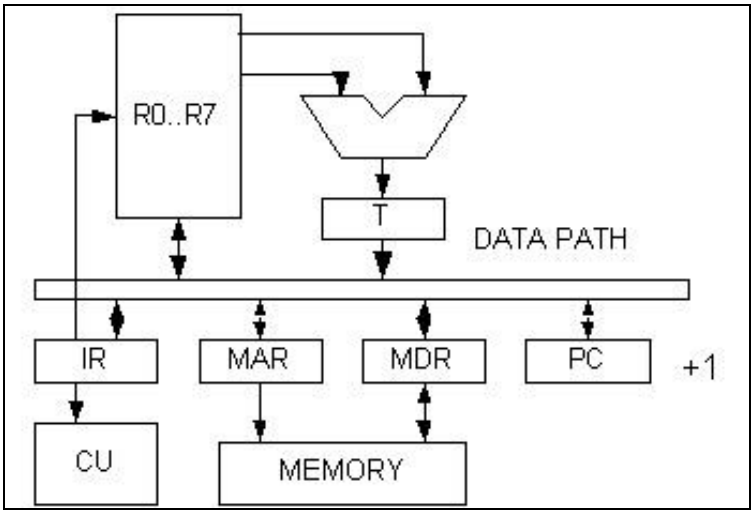

Figure 3: A Generic Computer Architecture

Table 4: Micro-Operations

Micro-Operation	MEMORY	R0..R7	MAR	MDR	Bus
T->Bus					T
Bus-> (MDR,MAR)	(MAR,MDR)				
Memory->MDR			Address	Memory	
Bus->R0..R7		Bus			
IR->Bus					IR

It is important to note how we model these micro-step tables. We generally assume an architectural transition/step will take one time cycle and each transition/step takes us to a potential new state in the state machine. The micro-steps related to an architectural step occur within the same time step. The microarchitecture steps do not represent state at this level. Instead, they perform a number of combinational logic functions to achieve the goals of the architectural step. In our example case, there were up to five micro-steps for a given architectural step. Following are the steps involved in modeling micro-operations using micro-transition tables.

1. Identify all the possible micro-operations.
2. Create a table mapping architectural action bits to the micro-operations list.
3. For each architectural role:
    a. Identify the minimal list of micro-operations possible at the first micro-step.
    b. Create a table linking the drivers and receivers of micro-operations.
    c. Repeat the same process for the second and third micro-steps if needed.
4. Create tables for each of the possible micro-steps for each of the architectural roles that enable information flow between microarchitecture components.

Table 4 shows a sample micro-operations table. The entries in the first rows are the receivers of the data from entries in their column. The second row shows that if the model sees "T->Bus" signal asserted, it will load ALU buffer T into Bus. Unlike role-based transition tables where only one transition table is active during one time slot, many micro-operations transition tables' transitions may execute sequentially inside one time slot. In our case, there are up to five tables following the transition table of each role. If an architectural transition required three micro-operations, the corresponding transitions in each of the three tables will be executed sequentially hence enforcing the order required for the micro-operations' data flow.

Atomic Actions and orderings: The architectural specification also had the following three challenging requirements:

1. The protocol requires ordering among write messages from the same agent.
2. There is no ordering requirement among different agents for a given instruction i.e. the interconnection fiber between different agents does not guarantee ordering among different agents.
3. An incoming forwarding message may take multiple cycles (transitions/steps) to complete and no other protocol transitions should happen during that time. The reason behind this is, a forwarding message has to be compared with all the waiting messages in a buffer. Each comparison will take one architectural transition/step. The maximum number of comparisons is only limited by the depth of the buffer. The agent will not honor any new messages during this process.

We modeled these requirements using FIFOs. Any new incoming write request for an agent is appended at the end of its FIFO. The outgoing requests are picked from the top of this FIFO. Each entry in the FIFO is assigned a unique transaction ID but the entry's addresses may not be unique.

To mimic the interconnection fiber and the out-of-order requirement, we inserted a buffer and an artificial architectural role/step between the agents and interconnection fiber. If this buffer is not full, an outgoing request from any agent lands into this buffer. When this artificial architectural step is executed, the interconnection buffer may pick randomly any valid entry from this buffer and hence models the actual out-of-order behavior.

This requirement was equivalent to having a macro-step (as compared to micro-steps above) and was implemented using a semaphore concept from the concurrent systems. We devised a global forward signal. All the architectural steps/roles were disabled when this global forward signal is asserted. The only exception was the forward message processing role/step of the active agents. When the active agent started processing a forwarding message, it asserted this global forward signal, disabling all other possible architectural steps. Once all the valid entries in the active agent's FIFO were processed, this global signal was de-asserted and at that point, any of the enabled roles could start executing.

Table 5: Stimuli Generation

Instruction	Requester	Server	CMD	Exclusive	Partial
12'b000000000000	Local	Local	Read	1'b0	1'b1
12'b000000000001	Local	Local	Read	1'b0	1'b0
12'b000000000010	Remote	Local	Write	1'b1	1'b1
12'b000000000011	Remote	Local	Write	1'b0	1'b0

Legal stimuli generation: An architecture executes a well-defined legal instruction set. An instruction consists of multiple fields. Each field may have a legal set of values. All the combinations of the values from the different fields may not form the legal instructions. In fact, the legal instruction set is only a small subset of this. One way to restrict input space is to constrain these legal combinations using assumptions. However, that process is error prone and hard to maintain. We worked with the architects to auto-generate a legal instruction set and used it as our stimuli. Still, there were thousands of possible instructions. Each legal instruction was assigned a unique ID from a 12-bit counter called "instruction". FV randomly chose a value from this counter to pick a legal instruction and used it as stimuli to our architectural model. The chosen instruction asserted proper command fields as well as configuration settings.

Table 5 shows a sample of our stimuli table. The first column in Table 5 assigns a unique number to each legal instruction. The rest of the columns define the instruction. For example, the first row defines a partial local read instruction that will be executed if "instruction" counter is randomly assigned a value of "12'b000000000000". The last row of the table defines an exclusive write instruction for a remote requester if the "instruction" counter gets a value of "12'b000000000011".

Unique transition identifiers: An important piece of the specification was the assignment of a unique identification (ID) number to each transition. It was a great facilitator in debugging, as any failure trace only needed to print these unique IDs. A Tcl script used this list of IDs with protocol tables to print a complete failure transaction in terms of the agents' transitions. It was a great help and saved an enormous time in debugging. These annotations also helped in simplifying sequence coverage, as described later.

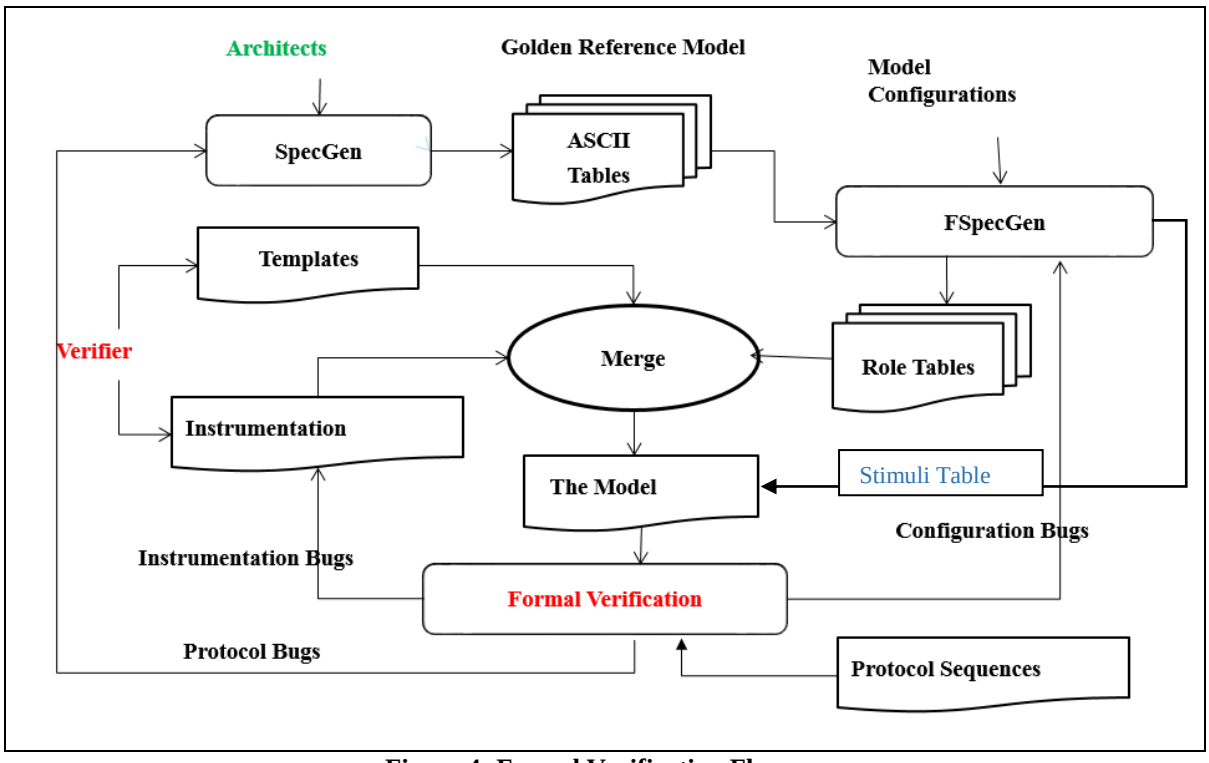

Figure 4: Formal Verification Flow

Formal Verification: A formal verification effort requires creation of a logically consistent model that captures the behavior of the design. It also needs a set of checks and coverage properties which capture the intent of the design in a declarative way [7]. Figure 4 describes our flow for the creation of the formal model and its formal verification. Further details on this flow can be found here [6]. The key addition to this flow is the stimuli generation from FSpecGen as discussed earlier. Once we have a model in place, the following steps are required to reach complete verification.

Path clearing: We define path clearing as checking if our model is alive and executing legal instructions. Here is a list of items we went through for path clearing.

1. We created a set of assumptions. It was important that these assumptions should not over-constrain or under constrain the model. We were generating our legal stimuli set using scripts. This reduced our assmuptions to a relatively small set. For such a big model, we had only around 10 assumptions. A few example assumptions are:
    a. Assuming address space is of size 2.
    b. Assuming agents' FIFO depth is 4.
    c. Assuming if an address is faulty, it can or cannot become fault-free.
    d. Assuming the depth of different channels.
    e. Assuming only a subset of instructions is available by constraining the "instruction" counter.
2. We ensured the reachibility of all the architectural transitions. This, in turn, validated that the assumptions made in step one were not over-constraining the model.
3. All the legal states were reachable.
4. We ensured reachibility of all the states of the role statemachines for each of the agents. We also validated that all the legal transitions of these role statemachines are executeable.

Functional verification: Following is a selected list of formally proven properties:

1. When there are no outstanding transactions the system only can be in one of the legal states.
2. The completeness of each architectural role.
3. Role statemachine verification.
4. Cache coherence is maintained.
5. Agents' messages' constraints are observed.
6. There are no deadlocks because of the interaction of the architectural components.
7. A dirty block in cache will always be written back to the memory before its invalidation.

We are providing here a sample property used for agents constraints' check. We used rigid variables[6] to randomly choose an agent, a valid write entry in it, and data coming with it. The "start_event" captures the moment when a chosen agent gets a write request and puts it into the designated entry into its FIFO. The "end_event" captures the moment when we are done with this transaction. We maintained a counter for outstanding write requests for each agent and that was passed as "outstanding" into this checker. This checker helped us in cleaning up the agent model and found a number of bugs in our modeling effort.

```
property agents_constraints(start_event, start_data, end_event, end_data, Outstanding, clk, rst);

    logic [$bits(start_data)-1:0] local_data;
    logic [$bits(Outstanding)-1:0] numAhead;
    (start_event, local_data = start_data,numAhead = Outstanding)
          ##1 (numAhead > 0 ##0 end_event[->1], numAhead--)[*]
                                          ##1 (numAhead == 0 ## 0 end_event[->1]) |-> end_data == local_data;
  endproperty
```

Functional coverage: We used graph-algorithms to generate all possible paths between the start and end states of each agent[3]. The discovered paths are used to generate protocol sequences. The formal model was used to find the missing sequences as well as to prove reachability (and/or unreachability) of these sequences. One key advantage of this effort was that each transition in the model had a unique identification number. We used a register for each agent to capture its latest transtion idenfication number and tracked these registers' transitions to evaluate the functional coverage of each agent. It simplified our coverage modeling greatly as compared to the previous effort[3]. Here is a sample of our protocol coverage sequences:

```
//A simple sequence to capture the current value of tad_nxm, the register that records Agents' transitions.
sequence tnxm(xnm);  (tad_nxm == xnm); Endsequence
//Auto generated sequences, while assuming default clocking and disable blocks are defined.
cover property((tnxm(20'h00000)[*1:$]##1 tnxm(20'h100a9)[*1:$]));
cover property((tnxm(20'h00000)[*1:$]##1 tnxm(20'h100aa)[*1:$]##1 tnxm(20'h2001a)[*1:$]));
cover property((tnxm(20'h00000)[*1:$]##1 tnxm(20'h1011b)[*1:$]));
cover property((tnxm(20'h00000)[*1:$]##1 tnxm(20'h101f3)[*1:$]##1 tnxm(20'h20019)[*1:$]));
```

Microarchitecture: Here is a selected list of the microarchitecture properties[4]:

1. The completeness of the microarchitectural table i.e. all the architetural transitions are completely supported at the microarchitecture level.
2. The data-flow at the microarchitecture level is not broken i.e. there is no data loss because of any missing or misplaced or misordered actions for any of the *architectural* transitions.

Data Consistency:

This property ensures that any read will get the latest write into the system. The modeling and verification of this property was a challenge as there were multiple write sources, partial-writes, fault cases, etc. We have proven a restricted form of data consistency properties in which after a random write instruction is executed in a randomly chosen legal state, we force a write back to the memory and check the incoming data against the memory data. We are presenting here a template of the data-checker used for this purpose[7].

```
//Data is captured at start_ev and will come out at the end_ev.
  property data_transfer(start_event, start_data, end_event, end_data, clk, rst );
    logic [$bits(start_data)-1:0] local_data;
    @(clk) disable iff(rst)
          (start_event, local_data = start_data) ##0
                                        (end_event or (!end_event ##1 (!start_ev throughout end_event[->1])))
           |-> (local_data == end_data);
  endproperty
```

Results and Conclusions: The complete architectural protocol and its associated microarchitecture actions were specified and verified using formal verification. We developed an innovative set of techniques and methods to handle this complex protocol. A number of bugs were found and fixed. Here is a generic list of the bugs found during this effort:

1. The first and biggest challenge is the completeness of the tables. There were missing transitions in home, remote, and IOB tables. The missing transitions cause dead-ends (a benign form of deadlocks).
2. The second biggest set of bugs originated from the bad transitions. The architect inadvertently inserted wrong transition of a few operations. For example, an ADD instruction is actually doing a MIN or MAX operation or has a wrong configuration of flags.
3. There were missing or wrong micro-operations.
4. There were few unreachable transitions as well. In this case, the architect thought a particular case could happen but it was not possible with the assumed architectural constraints.
5. Finally, once we have proven that all the transitions were reachable, we verified sequence coverage. We found that there were few sequences that were not reachable because of the architectural constraints. This enhanced our understanding of the architecture and forced a few modifications.

The verified tables were used as mapping functions in the RTL design. No protocol bugs escaped to RTL or later stages.

References:

[1] Weber, Ross. (2011) "Modeling and Verifying Cache-Coherent Protocols, VIP, and Designs". Jasper Design Automation, June 2011.

[2] Ikram, Shahid et al. (2014) "A Framework for Specifying, Modeling, Implementation, and Verification of SOC Protocols", September 2014, IEEE-SOCC, Las Vegas, Nevada.

[3] Ikram, Shahid et al. (2015) "Table-based Functional Coverage Management for SOC Protocols", March 2015, DVCON, San Jose, CA.

[4] Ikram, Shahid et al. (2016) “Formal verification of the microarchitectural features in the context of an architectural model”, November 2016, Jasper User Group Workshop 2017, Cupertino, CA.

[5] Ikram, Shahid et al. (2015) “Design and Verification of a Multichip Coherence Protocol”, March 2015, DVCON, San Jose, CA.

[6] Cerny, Eduard et al. (2015) “SVA: The Power of Assertions in System Verilog”, Second Edition, Springer, ISBN 978-3-319-07138-1.

[7] Seligman, Erik et al. (2015) “Formal Verification”, Morgan Kaufmann, ISBN 978-0-12-800727-3.