2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
OCTOBER 15-16, 2024

Automated Design Behaviour Extraction of SoC Interconnects Using Formal Property Verification

Jan Hahlbeck, NXP Semiconductors Germany GmbH (jan.hahlbeck@nxp.com)
Chandana G. P., NXP Semiconductors Germany GmbH (chandana.guddenahallipalaksha@nxp.com)
Görschwin Fey, Hamburg University of Technology (goerschwin.fey@tuhh.de)

***Abstract*—This paper describes an algorithm to extract the design behaviour of SoC interconnect modules by using formal traces in a fully automated flow. The main purpose of this flow is to unveil undocumented and unintended design behaviour which might harm security relevant aspects of the SoC. We rely on an abstract internal model of the SoC interconnect. Thus, no specification of the design behaviour is needed to run the abstraction flow. The results can be checked against the project specifications and requirements. By means of an example design of an AHB interconnect the algorithm gets demonstrated.**

Keywords—Formal Verification, Behaviour Extraction, Interconnect Verification, SoC Security

I. Introduction

As supplier of modern System-on-Chip (SoC) designs for radio and audio processing in the automotive domain, we deal with a variety of interconnect modules to implement the communication infrastructure between integrated hardware intellectual property (IP) blocks. The verification of interconnect modules is one of the crucial tasks in the SoC development cycle due to the high degree of freedom for possible communication channels in SoC interconnect IPs between managers and subordinates like depicted in Figure 1. The nature of interconnect modules is generic and they are configured based on project specific system requirements. These requirements include the number of managers and subordinates, the protocol interface type, access rights, error handling, performance and the underlying memory map. Interconnect modules are also highly security relevant to protect sensitive assets from unauthorized access [1]. We noticed that the total number of managers and subordinates per interconnect module increased over the years which results in a higher complexity of the underlying memory map and the associated verification effort. Typical interface types are industry standard protocols like AMBA AXI or AHB. Besides the growing complexity of the design itself, we identified the following challenges during the verification of interconnects:

- The provided documentation is outdated, e.g. not all memory map addresses are documented properly.
- Design engineers are not available to support verification efforts due to other priorities.
- Design architects created unintended design behaviour due to wrong interpretation of requirements.
- Design bugs or misconfiguration gets detected late in the project cycle as important verification tasks are executed on SoC level verification rather than on module level.
- In case the interconnect modules are third party IPs, the shipped verification suite is not sufficient to cover all project specific aspects especially when dealing with security relevant aspects [1].
- Aggressive time-to-market schedule makes it impossible to verify 100% of the design within the project schedule.

To overcome these issues we developed an algorithm to extract the interconnect design behaviour based on formal cover traces. Following the trend of an emerging usage of formal verification as methodology [2] and as part of our strategy to strengthen the usage of formal property verification (FPV) built on [3], we take advantage of the FPV ability to specify the destination rather than the full stimuli to get the Design-under-Test (DUT) into an expected target state. The mathematical exploration of the entire space of all reachable DUT states provides either a trace of how to reach the destination or vice versa proves the absence of a possible stimuli to get the DUT into the expected state [4]. The main focus of the presented algorithm is to unveil undocumented and unintended design behaviour which might harm security relevant aspects of the SoC. The DUT is intendedly handled as blackbox and besides a testbench toplevel and connected assertion-based verification IPs (ABVIPs), no further information is required. We are able to extract access rights, detailed subordinate accessibility information and a memory map by running a fully automated flow. The resulting memory map gets exhaustively verified by using FPV to ensure the validity of the extracted memory windows. All properties used for assumptions, covers and assertions are implemented by using System Verilog Assertions (SVA). As underlying formal tool we rely on Jasper FPV [5] and all scripting is done in Python.

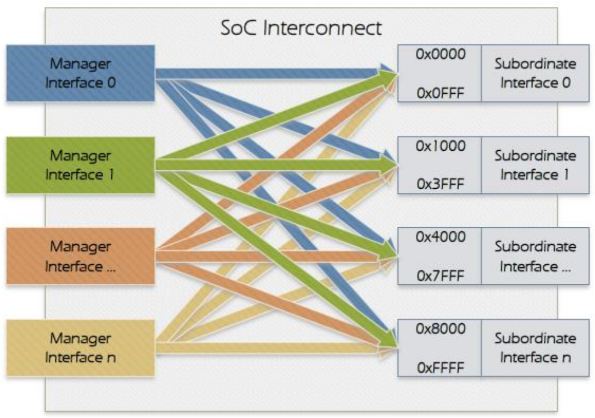


Figure 1: SoC interconnect architecture with mapped addresses

II. Proposed Algorithm

The proposed algorithm to extract the interconnect design behaviour is shown in Figure 2 and includes seven steps, divided in a low complexity part including basic checks and a more complex part for the memory map extraction and exhaustive verification. Prerequisites are the DUT and a testbench toplevel with connected ABVIPs for all standard protocol interfaces. ABVIPs include ready-to-use assumptions, cover properties and assertions to avoid any additional effort for standard protocols. Based on an abstract model, SVAs are generated to determine which read /write paths are useable in the IP core. Then, the memory mapping is probed with further assertions. The underlying formal proof ensures the exact extraction. All steps will be described in the next subsections.

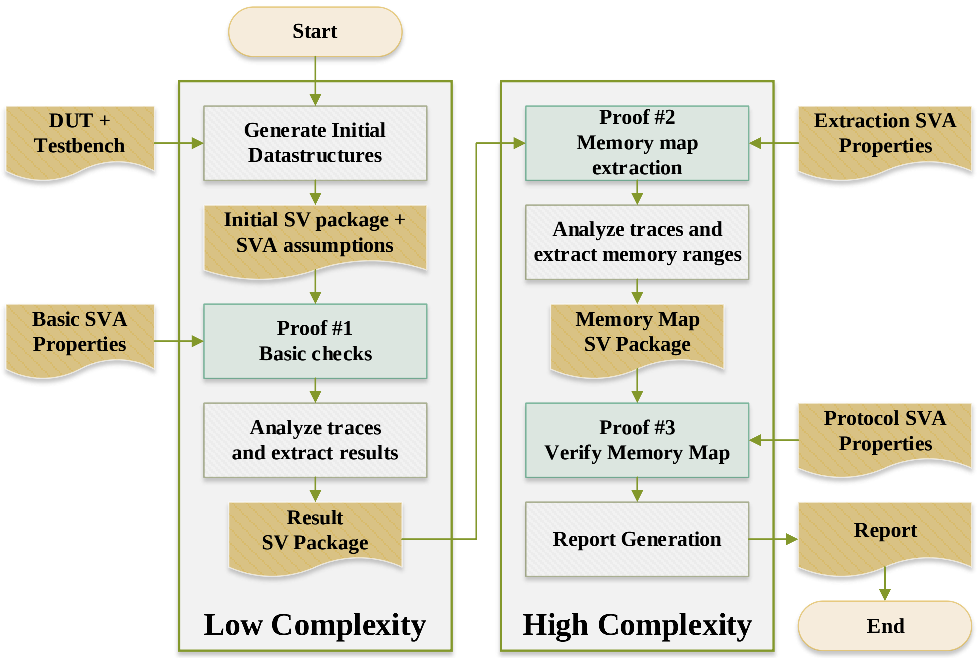

Start
DUT + Testbench
Generate Initial Datastructures
Initial SV package + SVA assumptions
Basic SVA Properties
Proof #1 Basic checks
Analyze traces and extract results
Result SV Package
Low Complexity
Proof #2 Memory map extraction
Extraction SVA Properties
Analyze traces and extract memory ranges
Memory Map SV Package
Proof #3 Verify Memory Map
Protocol SVA Properties
Report Generation
Report
End
High Complexity

Figure 2: Design behaviour extraction flow using FPV

Step 1: Generation of initial data structures and assumptions

In the first step a script parses the testbench toplevel including the DUT interface and identifies how many manager and subordinate ports are available at the DUT module boundary. Based on the extracted numbers, a script generates a SystemVerilog package which contains initial matrix structures including default assumptions that all managers can access all subordinates. This step is extended with the generation of protocol specific data structures, e.g. to store the PSEL, HREADY or TRANS characteristics for AMBA AHB [6]. In addition we generate an SVA module to apply data tagging on the write and read data ports of the extracted managers and subordinates to ensure unique data patterns. Data tagging is a widely applied FPV technique to use known data patterns on certain signals by using SVA assumptions. Figure 3 shows an example of how to tag the write data port of manager 0 with a unique value. This value can be detected for arrival in all of the subordinates.

```
// Assumption for Manager 0
// - No other manager is allowed to use wdata[3:0] = 4'h1.
// - Data [31:4] can be randomly used by formal tool
assume_manager_0_wdata : assume property (DUT.ahb_manager_wdata[0][3:0] == 4'h1);

// Cover property for Port 0, Subordinate 0
// Trace reachable   -> Manager 0 can     access Port 0, Subordinate 0
// Trace unreachable -> Manager 0 can NOT access Port 0, Subordinate 0
cover_wdata_manager_0_to_subordinate_0 : cover property (
  @(posedge clk) disable iff (!rstn)
    DUT.ahb_subordinate_psel[0][0] && DUT.ahb_subordinate_write[0] == AHB_WRITE
    ##1 DUT.ahb_subordinate_wdata[0][3:0] == 4'h1
);
```

Figure 3: Data tagging assumption and cover property example for AHB write data of manager 0 to subordinate 0

Step 2: First proof - Basic checks

The formal tool gets started for the first time to run a proof on basic checks like accessibility and protocol specific cover properties. This proof is based on the initial data structures in combination with unique patterns on write and read data ports of managers and subordinates. Previously implemented SVA cover properties are applied to each identified communication path by using a generate loop with respect to the extracted design constraints. Target of the first proof is to show the evidence or absence of formal traces to get data through the interconnect or to check e.g. if the AHB PSEL signal can be asserted, de-asserted and toggled. The cover property in Figure 3 checks whether the unique data from manager 0 can be observed in any of the subordinates.

Step 3: Analyze traces and extract results

In this step all covered and uncovered traces are analyzed by a Python logfile parser which checks the proof report of the formal tool. Based on these results the initial data structures get updated with the extracted behaviour from the traces of the first proof. All results are stored in a new SystemVerilog package, which can be picked up by the next processing step. Whenever a trace is available, it implies that a design behaviour is possible and the formal tools provides at least one evidence which input stimuli are required to reach the defined target state.

Step 4: Second proof - Memory map extraction

This second proof is again fully based on formal cover traces. With the help of dedicated cover properties and the already extracted design behaviour, it is possible to detect every start and end address of the interconnect memory map. We even support the extraction of multiple address windows per subordinate. Figure 4 shows how such a cover property looks like to extract a single address window. The principle to extract the start address is based on the assumption that there must be an address ADDR_START where the interconnect forwards a transaction from manager M to subordinate S, but when address ADDR_START-1 is used the transaction does not get forwarded. The same argument applies to the end of an address windows with ADDR_END and

ADDR_END+1 to extract for the end address of an address window. Whenever the formal tool is capable to generate such a trace, it contains information about the start and end address, which can be post-processed in the next step.

```
cover_address_map_single_window : cover property (
  @(posedge clk) disable iff (!rstn)
  // Cycle 0
  // - Transaction from manager m to subordinate s allowed.
  // - Formal tool can use any address.
  `AHB_WRITE_ADDR_PHASE_ALLOWED(m, s)
  ##1
  // Cycle 1
  // - Address is decremented by 1
  // - Data of previous transaction gets routed through correctly.
  // - Transaction from manager n to subordinate s restricted.
  DUT.ahb_manager_addr[m] == $past(DUT.ahb_manager_addr[m]) - 1 &&
  `AHB_WRITE_DATA_PHASE(m, s) &&
  `AHB_WRITE_ADDR_PHASE_RESTRICTED(m, s)
  ##1
  // Cycle 2 (Just set back address to initial address)
  DUT.ahb_manager_addr[m] == $past(DUT.ahb_manager_addr[m]) + 1
  ##1
  // Cycle 3
  // - Transaction from manager m to subordinate s allowed.
  // - Address is higher as previous address
  `AHB_WRITE_ADDR_PHASE_ALLOWED(m, s) &&
  DUT.ahb_manager_addr[m] > $past(DUT.ahb_manager_addr[m])
  ##1
  // Cycle 4
  // - Data of previous transaction gets routed through correctly.
  // - Transaction by manager m to subordinate p/s restricted.
  // - Address is incremented by 1
  DUT.ahb_manager_addr[m] == $past(DUT.ahb_manager_addr[m]) + 1
  `AHB_WRITE_DATA_PHASE(m, s) &&
  `WR_TRANSACTION_RESTRICTED(m, s)
);
```

Figure 4: Cover property example to extract a single address window

Step 5: Analyze memory map traces and extract memory map

All covered and uncovered traces of the memory map extraction proof are analyzed and post-processed in this step to detect if there is a single address window or multiple address windows available to access a certain subordinate. This is implemented by executing tool specific TCL commands to analyze formal traces.

Step 6: Third proof - Verify memory map

The extracted memory map gets fully verified with the help of already existing properties to verify all kinds of AHB transactions. For every address within the extracted ranges we check that read and write access is going through the interconnect according to the protocol specification. Based on the numbers of possible communication paths this step might take several hours to complete. We check e.g. that protocol specific signals like the AHB burst size get always routed through the interconnect to a subordinate whenever a valid address is used to initiate the AHB transaction.

Step 7: Report creation

The report creation is based on all previously collected data and writes out a single text based report which can be used by design and verification engineers to compare it against the interconnect specifications. A lightweight filter mechanism is used to detect bogus results by combining all extracted checks.

III. EXEMPLARY DESIGN UNDER TEST

The exemplary DUT is an AHB interconnect with a total number of 10 AHB managers and 11 AHB ports. Every port can handle up to two AHB subordinates. Figure 5 shows the DUT as part of the testbench toplevel. All interfaces are connected to AHB ABVIPs. Four SVA modules are instantiated to implement assumptions, cover properties and assertions. Splitting the basic checks, the memory map extraction and memory map verification allows to start separate proofs in the used FPV tool.

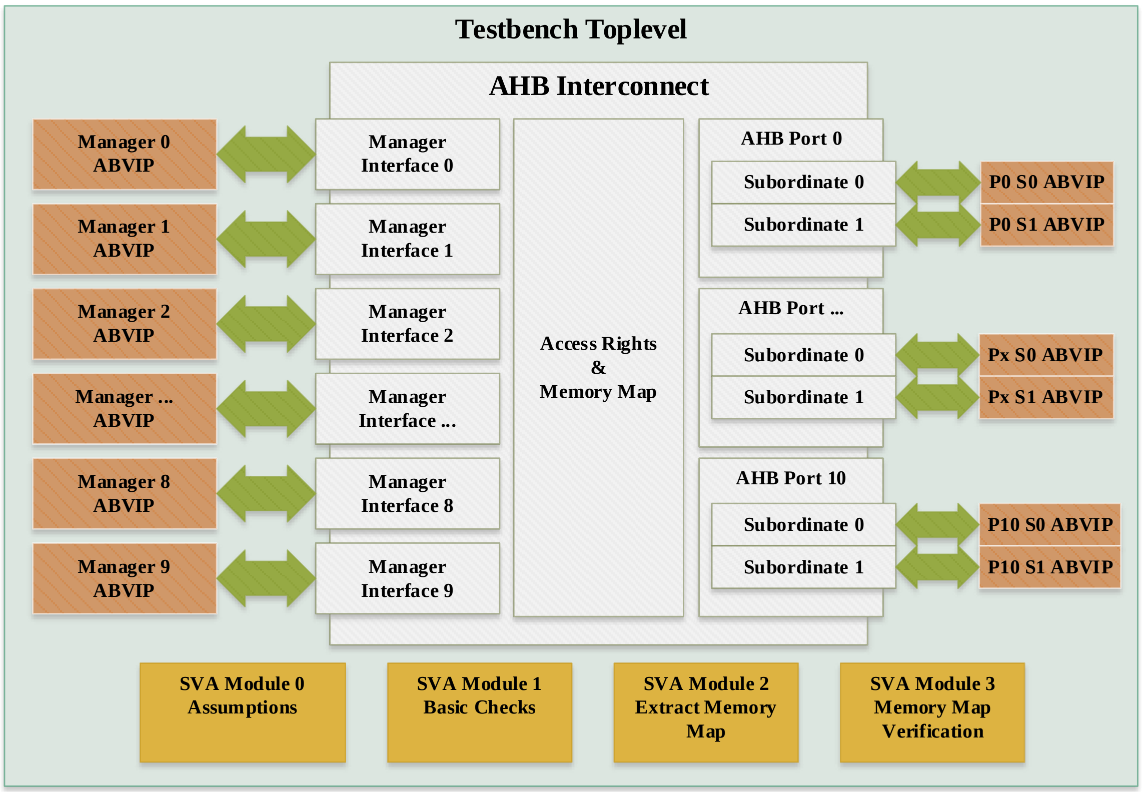
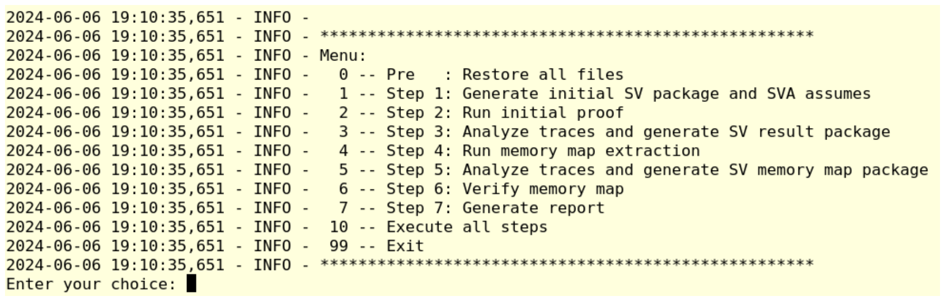


Figure 5: FPV testbench toplevel including DUT, ABVIPs and SVA modules

IV. RESULTS

A. Control Menu

The entire flow gets controlled by a Python script and can be started using a console menu depicted in Figure 6 either by executing it step by step for debugging or all steps at once. An additional option 0 got added to reset everything into the initial state.


```
2024-06-06 19:10:35,651 - INFO -
2024-06-06 19:10:35,651 - INFO - ****************************************************
2024-06-06 19:10:35,651 - INFO - Menu:
2024-06-06 19:10:35,651 - INFO -   0 -- Pre   : Restore all files
2024-06-06 19:10:35,651 - INFO -   1 -- Step 1: Generate initial SV package and SVA assumes
2024-06-06 19:10:35,651 - INFO -   2 -- Step 2: Run initial proof
2024-06-06 19:10:35,651 - INFO -   3 -- Step 3: Analyze traces and generate SV result package
2024-06-06 19:10:35,651 - INFO -   4 -- Step 4: Run memory map extraction
2024-06-06 19:10:35,651 - INFO -   5 -- Step 5: Analyze traces and generate SV memory map package
2024-06-06 19:10:35,651 - INFO -   6 -- Step 6: Verify memory map
2024-06-06 19:10:35,651 - INFO -   7 -- Step 7: Generate report
2024-06-06 19:10:35,651 - INFO -  10 -- Execute all steps
2024-06-06 19:10:35,651 - INFO -  99 -- Exit
2024-06-06 19:10:35,651 - INFO - ****************************************************
Enter your choice: █
```
Figure 6: Console menu to start the design behaviour extraction using FPV

B. *Number of Properties*

The total number of properties which gets used during the entire flow is shown in Table 1. The 21 assertions in the global assumption module are used to ensure that no other data tags are used, which might be introduced from somewhere inside the DUT. This ensures that the data originating at a specific manager or subordinate is unique. In addition the ABVIPs include almost 700 assumptions and 2000 cover properties to ensure that all transactions are compliant with the AHB standard. All numbers will dynamically change with respect to the DUT under consideration.

Table 1: Number of properties per SVA module

Module	Assumptions	Cover Properties	Assertions
Global Assumptions (Data Tagging)	32	-	21
Basic Checks	-	594	-
Memory Map Extraction	-	200	-
Memory Map Verification	14	20	330
AHB ABVIP Managers	390	820	10
AHB ABVIP Subordinates	286	1056	-
Total	722	2690	361

C. *Runtime*

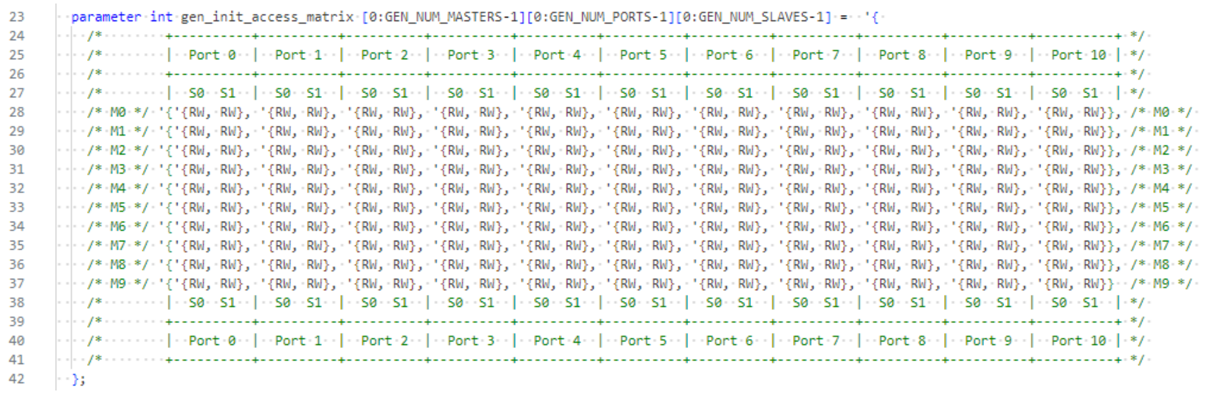
The total runtime of the flow for the DUT without re-using any cached data is around 3 hours like shown in Table 2. The basic checks are converging after just 3 minutes, as just low complexity cover properties are used. The memory map extraction takes around 15 minutes and the exhaustive memory map verification runs for 2:45 hours. The runtime of the third proof is highly related to the number of extracted address windows.

Table 2: Runtime of all proofs

Proof #	Runtime [h:mm]
Proof 1: Basic Checks	0:03
Proof 2: Memory Map Extraction	0:15
Proof 3: Memory Map Verification	2:45
Total	3:03

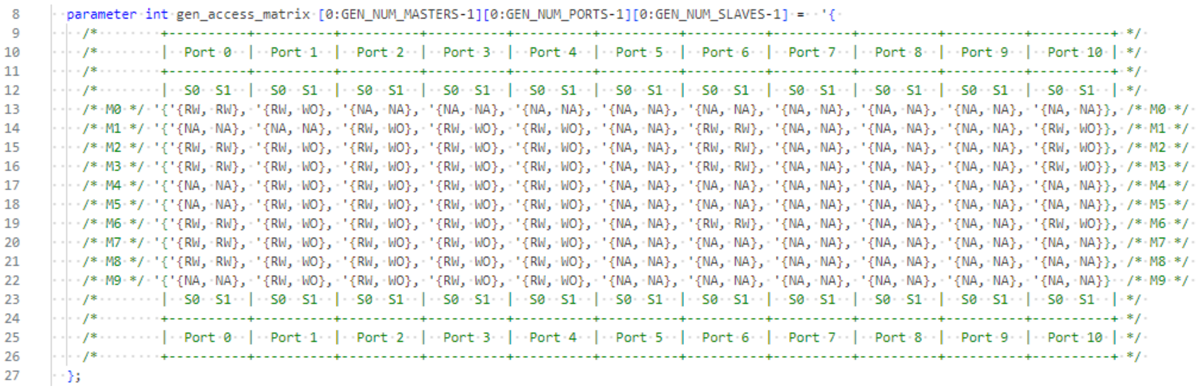
D. *Extracted Access Rights*

Initially the access rights are assumed to be read/write (RW) for all possible connections and they will be refined based on the results of the first formal proof. Figure 7 shows the initial assumption matrix which got generated for all possible connections. Figure 8 shows the updated matrix after the first proof got analyzed. Port P5, P7, P8 and P9 are not accessible at all. The only ports where the second subordinate is fully accessible are P0 and P6. Based on the automatically extracted information, that – due to underlying formal proofs – is fully correct, a designer can compactly assess whether this behaviour is intended or not. There is no manual effort required to gather these information.



```
parameter int gen_init_access_matrix [0:GEN_NUM_MASTERS-1][0:GEN_NUM_PORTS-1][0:GEN_NUM_SLAVES-1] =  '{
  /*         +----------+----------+----------+----------+----------+----------+----------+----------+----------+----------+----------+ */
  /*         |  Port 0  |  Port 1  |  Port 2  |  Port 3  |  Port 4  |  Port 5  |  Port 6  |  Port 7  |  Port 8  |  Port 9  |  Port 10 | */
  /*         +----------+----------+----------+----------+----------+----------+----------+----------+----------+----------+----------+ */
  /*         |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  | */
  /* M0 */ '{'{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}}, /* M0 */
  /* M1 */ '{'{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}}, /* M1 */
  /* M2 */ '{'{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}}, /* M2 */
  /* M3 */ '{'{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}}, /* M3 */
  /* M4 */ '{'{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}}, /* M4 */
  /* M5 */ '{'{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}}, /* M5 */
  /* M6 */ '{'{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}}, /* M6 */
  /* M7 */ '{'{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}}, /* M7 */
  /* M8 */ '{'{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}}, /* M8 */
  /* M9 */ '{'{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}, '{RW, RW}}  /* M9 */
  /*         |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  | */
  /*         +----------+----------+----------+----------+----------+----------+----------+----------+----------+----------+----------+ */
  /*         |  Port 0  |  Port 1  |  Port 2  |  Port 3  |  Port 4  |  Port 5  |  Port 6  |  Port 7  |  Port 8  |  Port 9  |  Port 10 | */
  /*         +----------+----------+----------+----------+----------+----------+----------+----------+----------+----------+----------+ */
};
```

Figure 7: Initially assumed access rights (RW=read/write)

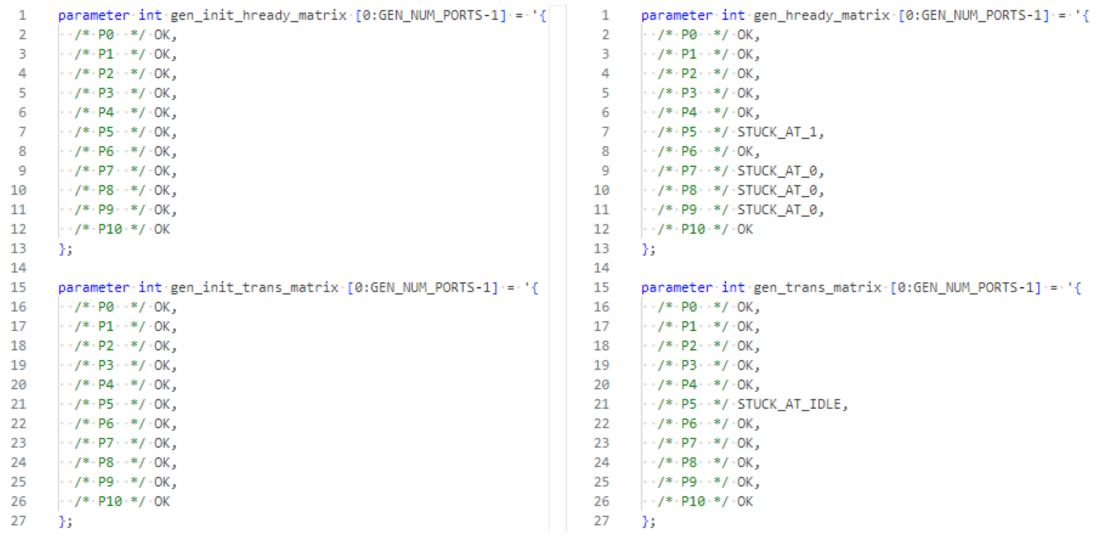


```
parameter int gen_access_matrix [0:GEN_NUM_MASTERS-1][0:GEN_NUM_PORTS-1][0:GEN_NUM_SLAVES-1] =  '{
   /*         +----------+----------+----------+----------+----------+----------+----------+----------+----------+----------+----------+ */
   /*         |  Port 0  |  Port 1  |  Port 2  |  Port 3  |  Port 4  |  Port 5  |  Port 6  |  Port 7  |  Port 8  |  Port 9  |  Port 10 | */
   /*         +----------+----------+----------+----------+----------+----------+----------+----------+----------+----------+----------+ */
   /*         |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  | */
   /* M0 */ '{'{RW, RW}, '{RW, WO}, '{NA, NA}, '{NA, NA}, '{NA, NA}, '{NA, NA}, '{NA, NA}, '{NA, NA}, '{NA, NA}, '{NA, NA}, '{NA, NA}}, /* M0 */
   /* M1 */ '{'{NA, NA}, '{NA, NA}, '{RW, WO}, '{RW, WO}, '{RW, WO}, '{NA, NA}, '{RW, RW}, '{NA, NA}, '{NA, NA}, '{NA, NA}, '{RW, WO}}, /* M1 */
   /* M2 */ '{'{RW, RW}, '{RW, WO}, '{RW, WO}, '{RW, WO}, '{RW, WO}, '{NA, NA}, '{RW, RW}, '{NA, NA}, '{NA, NA}, '{NA, NA}, '{RW, WO}}, /* M2 */
   /* M3 */ '{'{RW, RW}, '{RW, WO}, '{RW, WO}, '{RW, WO}, '{RW, WO}, '{NA, NA}, '{RW, RW}, '{NA, NA}, '{NA, NA}, '{NA, NA}, '{RW, WO}}, /* M3 */
   /* M4 */ '{'{NA, NA}, '{RW, WO}, '{RW, WO}, '{RW, WO}, '{RW, WO}, '{NA, NA}, '{NA, NA}, '{NA, NA}, '{NA, NA}, '{NA, NA}, '{NA, NA}}, /* M4 */
   /* M5 */ '{'{NA, NA}, '{RW, WO}, '{RW, WO}, '{RW, WO}, '{RW, WO}, '{NA, NA}, '{NA, NA}, '{NA, NA}, '{NA, NA}, '{NA, NA}, '{NA, NA}}, /* M5 */
   /* M6 */ '{'{RW, RW}, '{RW, WO}, '{RW, WO}, '{RW, WO}, '{RW, WO}, '{NA, NA}, '{RW, RW}, '{NA, NA}, '{NA, NA}, '{NA, NA}, '{RW, WO}}, /* M6 */
   /* M7 */ '{'{RW, RW}, '{RW, WO}, '{RW, WO}, '{RW, WO}, '{RW, WO}, '{NA, NA}, '{NA, NA}, '{NA, NA}, '{NA, NA}, '{NA, NA}, '{NA, NA}}, /* M7 */
   /* M8 */ '{'{RW, RW}, '{RW, WO}, '{RW, WO}, '{RW, WO}, '{RW, WO}, '{NA, NA}, '{NA, NA}, '{NA, NA}, '{NA, NA}, '{NA, NA}, '{NA, NA}}, /* M8 */
   /* M9 */ '{'{NA, NA}, '{RW, WO}, '{RW, WO}, '{RW, WO}, '{RW, WO}, '{NA, NA}, '{NA, NA}, '{NA, NA}, '{NA, NA}, '{NA, NA}, '{NA, NA}}  /* M9 */
   /*         |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  |  S0  S1  | */
   /*         +----------+----------+----------+----------+----------+----------+----------+----------+----------+----------+----------+ */
   /*         |  Port 0  |  Port 1  |  Port 2  |  Port 3  |  Port 4  |  Port 5  |  Port 6  |  Port 7  |  Port 8  |  Port 9  |  Port 10 | */
   /*         +----------+----------+----------+----------+----------+----------+----------+----------+----------+----------+----------+ */
};
```

Figure 8: Extracted access rights (RW=read/write, WO=write only, NA = not accessible)

E. *Extracted protocol specific behaviour*

As mentioned in the previous chapters we have implemented cover properties for protocol specific signals to gain more insights why certain communication paths are not accessible. Therefore, we selected three important AHB signals which are crucial to establish a communication between manager and subordinate: PSEL, HREADY and TRANS. Figure 9 exemplary shows the TRANS and HREADY behaviour by comparing the initial assumption with the extracted behaviour. Ports P7, P8 and P9 are not accessible because the HREADY signal gets stuck at 0, whereas P5 is not accessible due to a stuck at idle of the TRANS signal.



```
parameter int gen_init_hready_matrix [0:GEN_NUM_PORTS-1] = '{
  /* P0  */ OK,
  /* P1  */ OK,
  /* P2  */ OK,
  /* P3  */ OK,
  /* P4  */ OK,
  /* P5  */ OK,
  /* P6  */ OK,
  /* P7  */ OK,
  /* P8  */ OK,
  /* P9  */ OK,
  /* P10 */ OK
};

parameter int gen_init_trans_matrix [0:GEN_NUM_PORTS-1] = '{
  /* P0  */ OK,
  /* P1  */ OK,
  /* P2  */ OK,
  /* P3  */ OK,
  /* P4  */ OK,
  /* P5  */ OK,
  /* P6  */ OK,
  /* P7  */ OK,
  /* P8  */ OK,
  /* P9  */ OK,
  /* P10 */ OK
};
```

```
parameter int gen_hready_matrix [0:GEN_NUM_PORTS-1] = '{
  /* P0  */ OK,
  /* P1  */ OK,
  /* P2  */ OK,
  /* P3  */ OK,
  /* P4  */ OK,
  /* P5  */ STUCK_AT_1,
  /* P6  */ OK,
  /* P7  */ STUCK_AT_0,
  /* P8  */ STUCK_AT_0,
  /* P9  */ STUCK_AT_0,
  /* P10 */ OK
};

parameter int gen_trans_matrix [0:GEN_NUM_PORTS-1] = '{
  /* P0  */ OK,
  /* P1  */ OK,
  /* P2  */ OK,
  /* P3  */ OK,
  /* P4  */ OK,
  /* P5  */ STUCK_AT_IDLE,
  /* P6  */ OK,
  /* P7  */ OK,
  /* P8  */ OK,
  /* P9  */ OK,
  /* P10 */ OK
};
```

Figure 9: Extracted AHB TRANS / HREADY behaviour (initial assumption left, extracted results right)

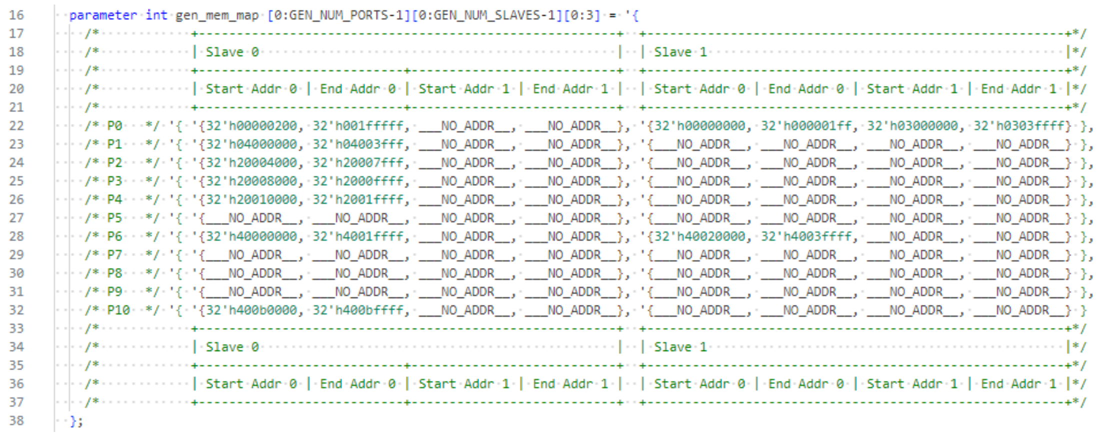
F. *Memory Map Extraction*

The memory map extraction gets executed only for connections where the basic checks ensure that the access is possible to optimize the runtime. Figure 10 shows the extracted memory map for all ports and subordinates. For most of the subordinates there exists just a single address window, only for port P0 and subordinate S1 there are two address windows. After passing this over to the design team of the interconnect IP we got the feedback that several unexpected issues are unveiled in the results which are not intended caused by wrong settings:

- The first address window of port P0, subordinate S0 with the address 32’h0000_0000 to 32’h0000_01FF is not expected and is a potential security risk, as it might expose registers / data which shall be not reachable.
- Port 5 is expected to be reachable, but a bug in the AHB TRANS logic prevents the access.

G. *Memory Map Verification*

The final memory map verification is based on assertions and checks that all extracted communication paths and address windows can be used to initiate a valid AHB write and read transaction. The final report contains a single line per check, for instance that the AHB write access to all subordinates of port P0 is possible. One design bug got discovered by these checks as the extracted address from any manager to port P0 subordinate S1 is not expected to have two address windows.



```
parameter int gen_mem_map [0:GEN_NUM_PORTS-1][0:GEN_NUM_SLAVES-1][0:3] = '{
    /*            +-----------------------------------------------------------+  +-----------------------------------------------------------+*/
    /*            | Slave 0                                                   |  | Slave 1                                                   |*/
    /*            +-----------------------------+-----------------------------+  +-----------------------------------------------------------+*/
    /*            | Start Addr 0 | End Addr 0 | Start Addr 1 | End Addr 1 |  | Start Addr 0 | End Addr 0 | Start Addr 1 | End Addr 1 |*/
    /*            +-----------------------------+-----------------------------+  +-----------------------------------------------------------+*/
    /* P0   */ '{ '{32'h00000200, 32'h001fffff, ___NO_ADDR__, ___NO_ADDR__}, '{32'h00000000, 32'h000001ff, 32'h03000000, 32'h0303ffff} },
    /* P1   */ '{ '{32'h04000000, 32'h04003fff, ___NO_ADDR__, ___NO_ADDR__}, '{___NO_ADDR__, ___NO_ADDR__, ___NO_ADDR__, ___NO_ADDR__} },
    /* P2   */ '{ '{32'h20004000, 32'h20007fff, ___NO_ADDR__, ___NO_ADDR__}, '{___NO_ADDR__, ___NO_ADDR__, ___NO_ADDR__, ___NO_ADDR__} },
    /* P3   */ '{ '{32'h20008000, 32'h2000ffff, ___NO_ADDR__, ___NO_ADDR__}, '{___NO_ADDR__, ___NO_ADDR__, ___NO_ADDR__, ___NO_ADDR__} },
    /* P4   */ '{ '{32'h20010000, 32'h2001ffff, ___NO_ADDR__, ___NO_ADDR__}, '{___NO_ADDR__, ___NO_ADDR__, ___NO_ADDR__, ___NO_ADDR__} },
    /* P5   */ '{ '{___NO_ADDR__, ___NO_ADDR__, ___NO_ADDR__, ___NO_ADDR__}, '{___NO_ADDR__, ___NO_ADDR__, ___NO_ADDR__, ___NO_ADDR__} },
    /* P6   */ '{ '{32'h40000000, 32'h4001ffff, ___NO_ADDR__, ___NO_ADDR__}, '{32'h40020000, 32'h4003ffff, ___NO_ADDR__, ___NO_ADDR__} },
    /* P7   */ '{ '{___NO_ADDR__, ___NO_ADDR__, ___NO_ADDR__, ___NO_ADDR__}, '{___NO_ADDR__, ___NO_ADDR__, ___NO_ADDR__, ___NO_ADDR__} },
    /* P8   */ '{ '{___NO_ADDR__, ___NO_ADDR__, ___NO_ADDR__, ___NO_ADDR__}, '{___NO_ADDR__, ___NO_ADDR__, ___NO_ADDR__, ___NO_ADDR__} },
    /* P9   */ '{ '{___NO_ADDR__, ___NO_ADDR__, ___NO_ADDR__, ___NO_ADDR__}, '{___NO_ADDR__, ___NO_ADDR__, ___NO_ADDR__, ___NO_ADDR__} },
    /* P10  */ '{ '{32'h400b0000, 32'h400bffff, ___NO_ADDR__, ___NO_ADDR__}, '{___NO_ADDR__, ___NO_ADDR__, ___NO_ADDR__, ___NO_ADDR__} }
    /*            +-----------------------------------------------------------+  +-----------------------------------------------------------+*/
    /*            | Slave 0                                                   |  | Slave 1                                                   |*/
    /*            +-----------------------------+-----------------------------+  +-----------------------------------------------------------+*/
    /*            | Start Addr 0 | End Addr 0 | Start Addr 1 | End Addr 1 |  | Start Addr 0 | End Addr 0 | Start Addr 1 | End Addr 1 |*/
    /*            +-----------------------------+-----------------------------+  +-----------------------------------------------------------+*/
};
```

Figure 10: Extracted memory map for all ports and subordinates (NO_ADDR is equal to 32’hFFFF_FFFF)

H. *Limitations*

The proposed approach is currently limited to AHB interfaces. Adapting it to additional standard protocol interfaces like AXI is under development. The number of possible address windows per subordinate is currently limited to 2, but can be scaled with a drawback of a runtime penalty. The total runtime might get critical for a higher number of managers, subordinates or accessible address windows as the time for formal proofs increases exponentially. For more complex designs a well-defined abstraction and black boxing strategy must be in place to deal with a possible state space explosion.

V. SUMMARY

In this paper we presented a fully automized flow to extract the design behaviour of an AHB interconnect by taking advantage of formal traces. The flow only requires the DUT and testbench toplevel as inputs. No specifications are needed. In seven steps we extracted the basic connectivity matrix, protocol specific behaviour and a full memory map. In a final step the resulting memory maps gets exhaustively verified to validate the extracted results. For the given example, it takes around 3 hours to execute the flow and multiple design bugs got unveiled. The flow can easily be re-started for new design releases. Based on the results we plan to strengthen the usage of FPV for design behaviour extraction on more complex designs.

VI. REFERENCES

[1] D. Y. H. P. M. Farimah Farahmandi, System-on-Chip Security, Springer, 2020.

[2] H. Foster, "Functional Verification Study," Wilson Research Group, 2022.

[3] Jan Hahlbeck, Steffen Löbel, Chandana G. P., "Towards a Hybrid Verification Environment for Signal Processing SoCs," in DVCon Europe, 2023.

[4] Erik Seligman, Tom Schubert, M.V. Achutha Kiran Kumar, Formal Verification: An Essential Toolkit for Modern VLSI Design, Morgan Kaufmann, 2023.

[5] Cadence, "Jasper FPV App," [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/formal-property-verification-app.html. [Accessed June 2024].

[6] ARM, "AMBA AHB Protocol Specification," 2021. [Online]. Available: https://developer.arm.com/documentation/ihi0033/latest/. [Accessed 26 03 2024].